DISSERTATION

Improving Intrusion Detection Systems

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Doktors der technischen Wissenschaften

unter der Leitung von

o. Univ.-Prof. Dipl.-Ing. Dr.techn. Mehdi Jazayeri
E184-1
Institut für Informationssysteme

eingereicht an der

Technischen Universität Wien
Fakultät für Technische Naturwissenschaften und Informatik

von

Dipl.-Ing. Thomas Toth
ttoth@infosys.tuwien.ac.at

Matrikelnummer: 9525588
Webergasse 15/29
A-1200 Wien, Österreich

Wien, im Mai 2003

Dipl.-Ing. Thomas Toth

Kurzfassung

Die vorliegende Dissertation beschäftigt sich mit der Lösung von drei wichtigen Problemen heutiger Intrusion Detection und Response Systeme.

1. Intrusion Detection Systeme müssen die beobachteten Aktionen mit einer sehr hohen Geschwindigkeit verarbeiten, um alle Attacken gegen ein System erkennen zu können. Gängige Intrusion Detection Systeme arbeiten die einzelnen Angriffsszenarien der Reihe nach ab — dies bedingt bei einer steigenden Anzahl von Szenarien eine immer geringer werdende Verarbeitungsgeschwindigkeit. Im ersten Teil dieser Dissertation präsentieren wir einen neuen Ansatz zur Verarbeitung von Aktionen. Dieser wendet Clustering-Algorithmen auf den zu entdeckenden Signaturen an, um effizient Gemeinsamkeiten von Szenarien zu entdecken, wodurch eine parallelisierte Verarbeitung der Szenarien möglich wird. Das Worst-Case Zeitverhalten der Sensoren kann dadurch deutlich verbessert werden wodurch auch die Resistenz gegen Denial-Of-Service Attacken gesteigert wird. Der Ansatz wird experimentell anhand des Open-Source Systems Snort gezeigt und evaluiert.

2. Im zweiten Teil wird das Konzept von Abstrakten Signaturen eingeführt. Damit soll das Problem von signaturbasierten Intrusion Detection Systemen gelöst werden, dass diese nur im vorhinein definierte Angriffe entdecken können, also unbekannte Attacken unentdeckt bleiben. Dies gilt auch für den Fall, dass die neuen Attacken Variationen von bereits bekannten Attacken sind. Dadurch haben Angreifer von Netzen gegenüber deren Verteidigern immer einen Vorteil. Abstrakte Signaturen charakterisieren ganze Klassen von Attacken indem die Gemeinsamkeiten aller Instanzen einer Angriffsklasse repräsentieren. Dies ermöglicht dem Intrusion Detection System zumindest Variationen von bekannten Attacken zu entdecken. Das Konzept von Abstrakten Signaturen von Attacken wird eingeführt und eine Methode zur effizienten Entwicklung dieser wird vorgestellt. Die Entwicklung, Realisierung und die Effektivität von zwei Abstrakten Signaturen wird besprochen und evaluiert.

3. Im dritten und letzten Teil dieser Dissertation wird eine Methode zur Abschätzung der Effekte von automatischen Verteidigungsmechanismen erläutert. Gängige Systeme verwenden nur sehr einfache Methoden um anhand von festgestellten Attacken adequate Verteidigungsmechanismen auszuwählen. Sie berücksichtigen dabei die Abhängigkeiten innerhalb eines Netzwerkes nicht. So kann es dazu kommen, dass durch Auswahl von ungeeigneten Verteidigungsmechanismen ein höherer Schaden entsteht als durch den Angreifer. Wir stellen eine Methode vor, mit der sich die Abhängigkeiten innerhalb eines Netzwerkes modellieren lassen, und mit der die Effekte von potentiellen Verteidigungsmechanismen simulieren und genau errechnen lassen. Dies ermöglicht den sicheren Einsatz von automatischen Verteidigungsmechanismen.

Abstract

This dissertation contributes to the solution of three important problems of current intrusion detection systems.

1. Intrusion detection systems have to process incoming events at a very high speed to be able to detect all attacks against a system. Current intrusion detection systems compare incoming events in a serial way with the signatures, which results in a performance decrease when more signatures are used for detection. In the first part of this dissertation we present a new of way of processing events which applies clustering algorithms for parallelizing the task of comparing events to signatures. The clustering algorithm is used for finding signatures with commonalities which are then used for constructing a decision tree, that can then be used for efficient event processing. The worst-case timing behavior can be improved using this approach which makes decision-tree based systems more resistant against denial-of-service attacks. The approach has been implemented and evaluated on the open-source intrusion detection system Snort.

2. In the second part of this dissertation the concept of abstract signatures is introduced. The concept of abstract signatures tackle the problem of signature-based intrusion detection systems that they cannot detect attacks that have not been modeled previously, i.e., for which a signature has been created. This is true even in the case that an attack is just a modification of an existing class of attacks. The defenders of networks therefore are always in disadvantage compared to attackers. Abstract signatures represent a whole attack-class by specifying the similarities of the instances of an attack- class. This enables abstract signatures to detect variations of existing attacks. The concept of abstract signatures is introduced and a method for an efficient development of abstract signatures is presented. The construction, implementation and evaluation of two abstract signatures is shown.

3. In the last part of this dissertation a method for evaluating the effects of automated intrusion response mechanisms is presented. Current intrusion response systems work on oversimplified data and use only simple methods to determine adequate response mechanisms when an intrusion is detected. The effects on the usability of the network cannot be calculated as the mission of a network is unspecified. This might lead to a situation where the cost of a chosen response mechanism is higher than the threat that is caused by an attack. We present a method which allows administrators of intrusion response systems to define the mission of a network and the dependencies within it. This information is then used for simulating the effects of response actions on the usability making it possible for the system to search for alternative responses that have a lower impact on the usability. This makes the use of automatic intrusion response safe.

Acknowledgments

This dissertation would not have been possible without the support of many people — I would like to thank all of them.

First of all I would like to thank Susanne Steiner for her patience and understanding. She gave me time for writing this dissertation, supported me in good and in bad times and kept many things away from me. Many thanks go to my parents and my relatives, who made my success possible by always providing an excellent environment for my education — they encouraged me never to give up and to reach for higher goals. I also would like to thank Susanne's parents for providing many things to us, that made it possible for us to concentrate on education and to have a carefree life.

I would also like to thank my colleagues at the Distributed Systems Group for their fruitful discussions and their support while doing research at this department. Special thanks to Prof. Wolfgang Kastner, Clemens Kerer and Dr. Engin Kirda who proof-read this dissertation and made valuable comments. Prof. Mehdi Jazayeri guided me during my life as a Ph.D. student and I want to thank him for this. He provided the necessary environment, he gave me some insight into teaching and allowed me to develop managing skills by working on a real-world project.

Special thanks go to Christopher Krügel. On the one hand we together experienced the feeling of having to rescue a real-world project, doing research in a previously unknown area (learning it the hard way) and working through whole nights. On the other hand we have been rewarded with success and with many unforgettable positive moments like the famous 'elevator-incident', which I don't want to miss.

This work has been supported in part by the European Community under the Information Societies Technology Programme IST-12637 and by the 'Fonds zur Förderung der wissenschaftlichen Forschung (FWF)' under the project OPELIX (P13731-MAT).

Contents

List of Figures

List of Tables

Chapter 1

Introduction

> The beginning of knowledge is the discovery of something we do not understand.
>
> Frank Herbert (1920 - 1986)

During the last decades a steady increase of computing power together with a decreasing cost of computing equipment was observed [51] [67]. Today not only large companies can afford running computers, but everybody can do so. Computers have become a general use tool. In order to work with these computers more efficiently and conveniently, computers have been interconnected using *networks*. One of the largest available networks, the Internet, now enables a person working at one computer to exchange messages with any other computer that is also attached to the Internet. Users working on Internet-connected hosts benefit from a large number of convenient applications such as the World-Wide Web (WWW) and e-mail. The vast potential computer users are provided with not only brings advantages but also disadvantages. The days in which computer security was equal to controlling physical access to machines are history. In those days, it was a requirement to have physical access to the computer that was to be used. This access was denied to anyone unauthorized. As typing in the commands directly at the computer console was the only method of launching programs in the early days, chances to take control of a machine were very low when physical access was denied.

Nowadays, the situation is completely different as more and more computers are attached to the Internet to benefit from the services the Internet provides. Servers run processes listening on defined network ports to capture requests sent by clients. The requests are processed and perhaps a reply message is sent back to the originator of the request message. By accepting messages from the network, the processes open a new way of control into servers. Even if a server is physically perfectly secured, this does not mean that there are no remaining security risks.

According to [10], networks are risky for at least three major reasons.

- First, and most obvious, more points now exist from which an attack can be launched. Someone who cannot get to your computer cannot attack it; by adding more connection mechanisms for legitimate users, also more vulnerabilities are added.

- A second reason is that the physical perimeter of your computer system has been extended. In a simple computer everything is within one box. The CPU can fetch authentication data from memory, secure in the knowledge that no enemy can tamper with it or spy on it. Traditional mechanisms — mode bits, memory protection, and the like — can safeguard critical areas. This is not the case in a network. Messages received may be of uncertain provenance; messages sent are often exposed to all other systems on the net. Clearly, more security-awareness is needed.

- The third reason is more subtle, and deals with an essential distinction between an ordinary dial-up modem and a network. Modems, in general, offer one service, typically the ability to log in. When you connect, you are greeted with a login or Username prompt; the ability to do other things, such as sending mail, is mediated through this single choke point. There may be vulnerabilities in the *login* service, but it is a single service, and a comparatively simple one. Networked computers, on the other hand, offer many services: login, file transfer, remote file access, remote execution, phone book, system status, etc. Thus more points are in need of protection — points that are more complex and more difficult to protect.

  Typical Internet-enabled hosts run many more services than single hosts. These services, run by *daemons*, implement the required standards for enabling the communication among hosts with different architectures and operating systems as well as arbitrary functionality such as Webservers, FTP-Servers etc. Daemons suffering from vulnerabilities can be exploited — resulting in unauthorized persons getting privileged access to these hosts. Equipped with privileged permissions a user can steal information or perform malicious activity — which is clearly undesired and has to be avoided.

The standard mechanisms for preventing exploits are firewalls and encryption. Firewalls can be seen as a network equivalent to restricting physical access to a computer. Only messages of authorized hosts are allowed to pass through a firewall. Encryption on the other hand transforms the messages into some unreadable form before it is sent away. During the transmission through the network the message is kept in this unreadable form, and only the recipient of the message has the knowledge of how to decrypt the message. These two protection mechanisms help to enhance the security in networks, but they are not a panacea. Some services such as HTTP servers or SMTP servers have to be publicly available (the firewalls lets all packets to these services pass) to make the Internet useful. Strict runtime requirements for typical public services also make the use of encryption prohibitive, as it would make message transfer slow. As a consequence, these services are popular targets for attacks.

1.1 The need for Intrusion Detection

In the following a typical setup of an Internet-enabled site is presented. Assume that a hospital is connected to the Internet, as shown in Figure 1.1. The hospital wants to pro-

vide some general information via the Webserver www and allows people to send e-mails to their Mail server mail, which is also used for transferring medical data to external physicians. Several stations are connected to the hospital's network which not only includes the treatment stations but also administration, billing etc. The setup guarantees that all information about patients is permanently available from every station to fertilize efficient treatment and administration. The hospital's network is protected by a firewall that guarantees that connections from the public Internet can only proceed to mail, www or the DNS server dns. The maintainers of the hospital's network assume that this setup provides good security, as only very few services are reachable from the Internet.

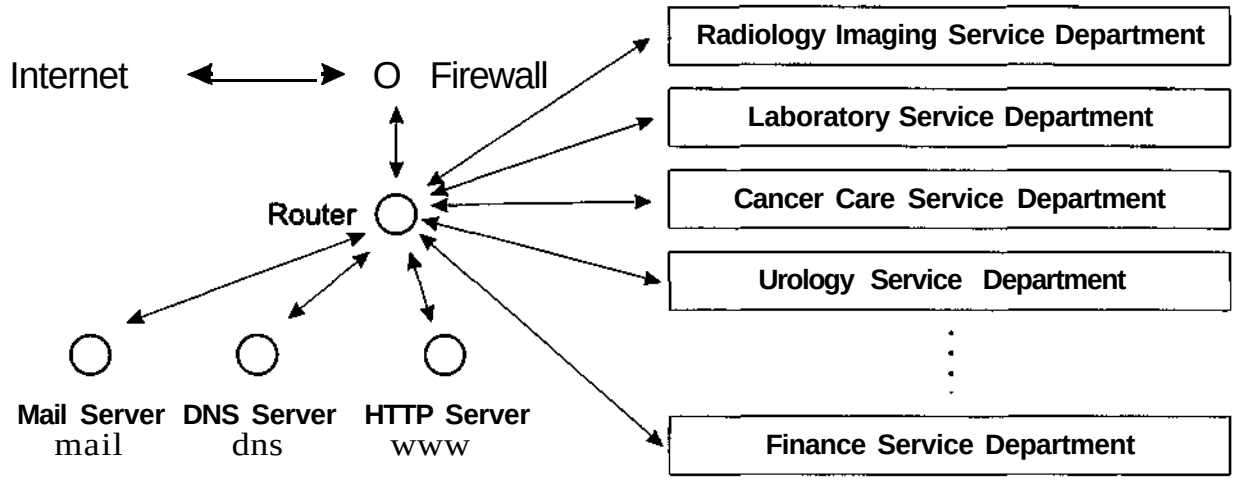


Figure 1.1: A Top View on a Hospital's Network Installation

In reality, the security situation is different. Although only the servers dns, www and mail are reachable from outside the hospital, the hospital's network has to face the threat of *hackers* — individuals that aim to get unauthorized access to systems. Usually hackers constantly look for new locations where they can launch their attacks from to masquerade their real identity, so the hospital is a possible target. Before a hacker actually breaks into a site, he first collects as much information as possible that might be relevant for the break-in. Thus he uses the (typical) publicly available information sources such as public databases to find out other existing machines in the victim's network. In our case, he uses tools such as nslookup and dig [28] to perform DNS zone transfers to find out valid machines within this network. He also finds out extra information such as the well known services at these hosts (a DNS server at dns, a Webserver at www and a mail server at mail), and the administrator's contact information. Using this information, he can start investigating the individual services in detail by monitoring the reply messages of requests sent to the services. The hacker finds out that mail is running CMail in version 2.4.9 on a Windows 98 machine by getting reply messages that are characteristic for this installation. By doing searches in the World Wide Web and in security related mailing lists, he finds out that this specific version of CMail is vulnerable [11] to a remote buffer overflow exploit which he is able to get. He launches the exploit against the vulnerable service and gets a command

shell with administrator permissions which he then uses to install a backdoor, i.e., software that enables him to get back into this machine without having to exploit the CMail server again. He does this just in case the vulnerable CMail server is replaced with a patched one. Subsequently he covers the tracks at the mail server by deleting lines from the log files so that a detection of the intrusion is unlikely. Then the attacker installs some extra software called *network sniffers* that listens on the network and tries to catch messages containing passwords that are sent in plaintext over the network. The attacker could use these passwords for accessing the resources on other hosts. The attacker is now behind the firewall, which has been the only entity limiting the type of traffic to the hospital's hosts. He can start scans within the hospital's net or use the hospital's mail server as a steppingstone to hack other networks. He can identify the services running on the other machines and possibly exploit them to get privileged access there, potentially finding the server that stores critical treatment data of patients. He can browse this information and even can alter it if this data is not secured (e.g., encrypted or hashed) — causing patients to be medicated in a wrong/harmful way.

In this case, the firewall did not help to keep the attacker outside the network. Even if CMail utilized encryption to protect the messages worked. CMail would have decrypted the message, and while handling it while they are transferred over the Internet, this exploit would have in a faulty routine, a buffer overflow would have occurred that causes the attacker's exploit code to be executed.

This example showed that the traditional security measures failed to keep an attacker outside the network. This can even result in a threat for human life, as shown in the above example and in [10] where a hacker shut down a system responsible for evaluating satellite image data. A storm was overlooked and people failed to recall a ship into its harbor, causing the ship's crew to be killed.

Intrusion detection (ID) systems are complementary to the traditional security mechanisms — they try to detect incidents in which systems have been intruded by analyzing the information that can be gathered from the system. In contrast to the above depicted single point security, where security checks are only performed at a single point, ID systems (IDS) are part of the *Defense-in-depth* approach [50]. In this approach a series of mechanisms (at different levels) are used, such that if one mechanism fails, another will already be in place to thwart an attack. The sensors of an ID system analyze the gathered data and determine whether an intrusion has occurred.

IDS can be categorized by various attributes. The most common attributes used are the location where data is collected and how the search for intrusions within the collected data is performed. Using the data collection location as discriminating feature, one distinguishes *host-based intrusion detection* (HID) from *network-based intrusion detection* (NID). HID systems collect their data directly from the resources that are available on the host, e.g., they make use of syslog messages, log files from applications, they monitor the behavior of users and even trace the system call behavior of processes. NID systems collect their data directly from the network by looking at the packets that are being transferred. Depending on the type of information gathering different results can be produced, because several events are only visible in one of the two domains. Categorizing IDS by the data processing

methods to find out whether a given data set contains an intrusion, one discriminates *misuse-based IDS* and *anomaly-based IDS*. Misuse-based IDS can be best compared to string pattern matchers. They find intrusions by analyzing the content of packets and searching for special keywords. Anomaly-based IDS can be best described as statistical engines. They compare the observed activities to a predefined profile — if an observed behavior deviates too much from the stored profile, then this behavior is assumed to comprise an intrusion.

The above depicted break-in into the hospital's network could have been prevented using IDS. First of all, the CMail server could have included a buffer overflow detection routine, a check that determines whether a request contains a buffer overflow. In case such a malicious payload is detected, the request would be simply dropped and prevented to get into the vulnerable parts of CMail. In this case, the break-in could have been prevented provided that there is no other vulnerability.

Even if the attacker manages to get into the network, his further activity is detectable by ID systems. The scans of the attacker that are aimed at detecting new hosts and vulnerable services can be detected with current network-based IDS within a few seconds. Even the fact that the attacker runs additional software that captures all the network packets to gather some username/password pairs causes alarms — the sniffer switches the network device of mail into the *promiscuous mode* — which is improbable for a regular node. A host-based IDS also detects the installation of the backdoor at mail — any changes on files are detected immediately, as the cryptographic checksums of files change and new files can be easily detected. Host-based ID systems also monitor the commands the attacker issues, which is completely different from the commands the system administrator uses at mail.

The various alerts generated by the ID system are then used to combat the intrusion at a very early stage — the local system administrator is informed via e-mail, SMS or some other means. The attacker can be logged out automatically, the compromised user account can be closed down immediately, mail can be put into a survival mode in which a login is only possible directly from the console, critical files can be restored from a backup system and unknown processes are killed. The firewall can be automatically reconfigured in such a way that only connections from the various stations are accepted, but not from the outside. All this leads to the situation in which the system is moved into a secure state, the system administrator is informed about this security incident is given time to fix the problems, preventing the attacker from penetrating the complete network.

1.2 The Vision — improved Intrusion Detection

Intrusion detection systems are a good way to improve the security level of the system as they look for hints that betray the hacker. Unfortunately also hackers have too access to IDS and can analyze their workings. They can find out which actions an ID system considers as 'evil' and which weaknesses ID systems have. Knowing the actions that reveal their existance in a network, the hacker from the above example could have made his scan on the network only slowly — in this case an ID system that only observes a finite time

window could not have detected the attacker's presence.

A problem of most ID sensors, especially of network-based ones, is that they cannot keep up with the speed events occur on regular computers if they have to search for a large number of signatures. Missing a single network packet can lead to the problem of being unable to catch intrusions — if packets containing the patterns the ID system searches for are dropped because of full queues, the ID system fails to identify the attack. Usually the signatures are compared one by one. This serialized method of search is very inefficient, as the processing time of events increases proportional with the number of used signatures.

In addition IDS have some major flaws that reduce their usefulness. First of all most sensors have the problem of either missing attacks *(false negative)* or detecting attacks where in fact there are none *(false positive).* Misuse-based IDS usually have a very low false positive rate while having a higher false negative rate, i.e., being unable to identify previously unknown intrusions. Anomaly-based IDS have the big advantage of being able to identify previously unknown attacks (compared to misuse-based IDS) as they do not rely on databases with exact signatures; on the other hand they have a frightening high false positive rate. If a system administrator is confronted with false positives from an anomaly-based IDS every minute, he will soon cease to use it since he would have to watch the generated alerts continously, preventing him from doing other useful work. So the gain of IDS is voided by bad detection behavior.

The automatic response component determines upon the reported alerts generated by an Intrusion detection system how to react upon this attack. The aim of the reaction is to limit damage the attacker can cause by isolating the intruder or moving him out of the system. Usually IDS use very primitive mechanisms such as static mapping tables for determining which action to launch in response to an identified intrusion. These are inherently inflexible mechanisms and do not take the mission of a network into account. This can, as will be shown later, lead to situations where the cost of automated responses can be much higher than their benefits. Many managers of companies running publicly available services are afraid of enabling these mechanisms, as being unavailable for the customers can result in loss of reputation and money.

In the following the goals of this dissertation are enumerated.

1. We show how a sophisticated search for the signatures can help to make network/misuse-based systems more scalable with respect to the number of signatures that are searched for.

2. We improve signature-based detection. We introduce the concept of *abstract signatures* which are signature-based detectors with both a low false positive and false negative detection rates. Instead of having exact signatures (which have to be updated regularly), an abstract signature is used. Abstract signatures describes a whole class of attacks and therefore is equivalent to a collection of all possible signature instances of an attack- class. For attacks where such an abstract signature can be found, it is not necessary to update the signatures that describe these attacks. By being able to reduce the number of used signatures, abstract signatures also help mitigating the performance problems.

3. We make the use of automatic intrusion response safe. We show that response impact evaluation can prevent self denial-of-service caused by unsophisticated intrusion response mechanisms. This allows operators of services to take advantage of self defending networks without having to worry about launching counterproductive responses.

1.3 Contribution

This dissertation tackles the three above mentioned problems of intrusion detection. The following list describes the contribution of this dissertation.

- The design, implementation and evaluation of a network-based ID system which has been enhanced with automatic rule clustering as well as multiple pattern matching to reduce the per-event matching overhead.
- The concept of of *abstract signatures* is introduced which allows covering a whole class of signature instances. The design, implementation and evaluation of two samples of abstract signatures are given.
  - A host-based sensor for detecting buffer overflow exploits in typical request oriented services
  - A network-based sensor for detecting worms in local networks
- The design, implementation and evaluation of a response subsystem including an impact evaluation component that allows to specify the mission and the dependencies of/within a network and to calculate the cost of response actions.

1.4 Organization

This dissertation is organized as follows:

Chapter 2 deals with the related work of this dissertation. Chapter 3 presents a way of making signature-based intrusion detection systems more scalable with respect to the number of rules using decision trees. As a case study the theoretical performance properties of Snort are investigated and it is shown that the applied data structure causes the dramatic performance impact of Snort's detection engine when many signatures are used. We then present the design, implementation and evaluation of Snort-NG. Snort-NG applies clustering algorithms to create a data structure that allows parallel signature searches, changing the performance behavior of the detection engine. Chapter 4 then describes the concept of *Abstract Signatures* as well as two sensors implementing abstract signatures. A host-based sensor is presented that can detect buffer overflow exploits in service requests by analyzing the executability of the payload. A network-based sensor is presented that is able to detect

worms by analyzing the connection patterns among hosts. Chapter 5 describes the design and implementation of a response subsystem that allows the definition of a network's mission that can be used for issuing responses with minimum negative impact.

Chapters 3, 4 and 5 each contain an evaluation of the presented work. Finally, Chapter 6 evaluates the work of this dissertation as a whole, concludes and outlines further research possibilities.

Chapter 2

Related Work

> The important thing is not to stop questioning. Curiosity has its own reason for existing. One cannot help but be in awe when he contemplates the mysteries of eternity, of life, of the marvelous structure of reality. It is enough if one tries merely to comprehend a little of this mystery every day. Never lose a holy curiosity.
>
> Albert Einstein (1879 — 1955)

The intent of this chapter is to present the related work of this dissertation. The first part gives a brief overview of intrusion detection, clarifies common terms and shows a classification of existing intrusion detection systems to make clear to which area contributions have been made.

The second part deals with signature-based IDS. It shows features and methods employed by currently available systems and explains the common problems of these systems. Furthermore it shows that nearly all deployed systems suffer from *performance problems* and the problem of being *unable to detect new attacks.*

The third part of this chapter deals with problems that arise when an automatic response system is coupled with an intrusion detection system. In this case the response system automatically launches countermeasures when an intrusion is detected. We show that this can lead to undesirable situations where the use of the underlying network is voided by imprudent countermeasures.

2.1 Intrusion Detection Overview

This section gives a brief introduction into intrusion detection systems, explains common terms and used concepts and shows a classification of existing approaches.

At first the terms *intrusion, intrusion detection* and *intrusion detection system* are introduced, taking definitions from [4].

Definition:

An *intrusion* is a sequence of related actions by a malicious adversary that results in the occurrence of unauthorized security threats to a target computing or network domain. An intrusion consists of a number of related steps performed by the intruder that violate a given security policy. The existence of a security policy that states which actions are considered malicious and should be prevented is a key requisite for an intrusion. Violations can only be detected, when actions can be compared against given rules.

Definition:

Intrusion detection (ID) is the process of identifying and responding to malicious activities targeted at computing and network resources. This definition introduces the notion of intrusion detection as a process, which involves technology, people and tools. An *intrusion detection system* is a computer that performs intrusion detection.

As insinuated in the introduction, intrusion detection systems are classified upon their event processing. Misuse based intrusion detection uses an a-priori knowledge of activities that together form an attack, while anomaly based intrusion detection detects intrusions by comparing actual behavior to a profile, which has previously been generated. According to [43] and [75] intrusion detection can be classified as shown in Figure 2.1.

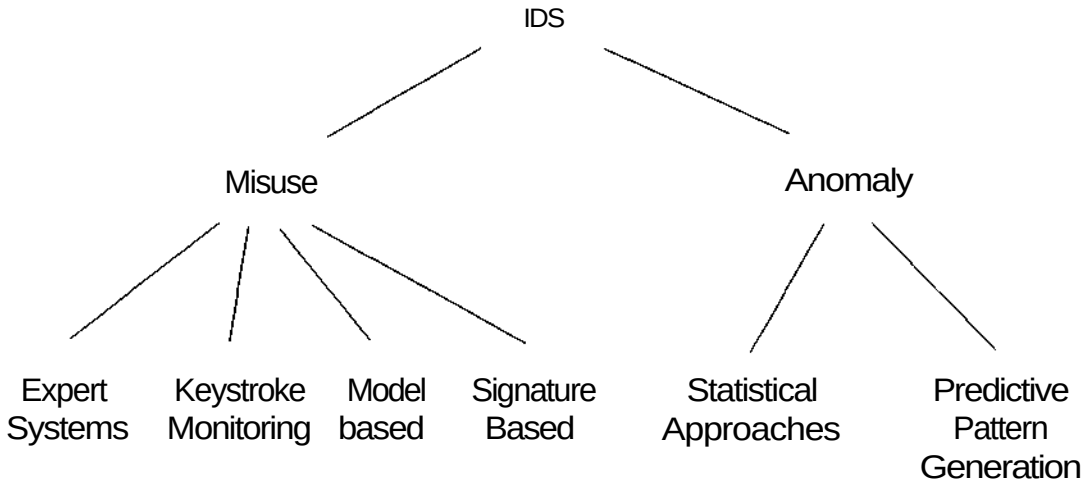


Figure 2.1: Classification of IDS

The individual types are explained in short:

- **Expert** Systems such as [5] and [59] contain knowledge of attacks as 'if-then' rules [32]. Each attack consists of several rules, each having a left and a right side. The left side contains a condition (the *if*-part). This condition is tested against the current situation and if the condition is met, the right side (the *then*-part) is executed,

possibly activating other rules. When the conditions of all rules of an attack are activated, this attack is identified. The biggest problem of expert systems is that they do not support a sequential ordering among the individual events, because the individual conditions of rules are not recognized to be ordered.

- **Model Based Intrusion Detection** which was presented in [23] combines models of misuse with evidential reasoning to support conclusions about an attack's occurrence. The basic statement of model based intrusion detection is that certain attacks can be inferred by other observable activities. The attack scenarios are constructed from a sequence of behaviors; all attack scenarios are stored in an attack database. In each situation the system considers a subset of these attacks with a certain probability as likely to be performed. To verify this assumption, the audit trails are searched for audit-records that support of refute this assumption. One component, the *anticipator* determines which behavior has to be searched for in the audit trail and hands this information to the *planner.* Then planner determines how this behavior will show up in audit records and transforms this into a system dependent audit trail match. The attacks that are considered to be active are updated each time new evidence for scenarios comes up. The likely-hood of attacks increases if more evidence accumulates and decreases for all others.

- **Keystroke Monitoring** monitors the keystrokes of users to determine whether an attack is ongoing. It searches for sequences of characters that together indicate an attack. Unfortunately the attacker can easily elude the system by exchanging the commands.

- **Signature Detection** can be best compared to pattern matching. A set of predefined signatures is used to determine which events from an event stream show some suspicious/intrusive behavior. The exact signature are formulated in such a way that they can be directly be used for searching within the event stream.

- **Statistical approaches** generate statistical measures to determine how far the observed behavior deviates from the previously measured one. Activity measures like the consumed CPU time, the consumed network bandwidth and the number of service invocations are typically taken as measures. Usually several of these measures are included in a profile. Some systems merge the currently measured profile with the stored one, while others keep the profile constant for a certain amount of time. The major disadvantage of statistical approaches lies in the fact that an attacker can gradually train a system until illegal behavior is considered to be legal. It is also not obvious which measures to include into the profile.

- **Predictive Pattern Generation** tries to predict the future events upon the events that already occurred. If the next observed event has a low probability this indicates a deviation of the existing profile. Several ways realizing predictive pattern generation exist, such as Bayesian networks [79], Markov-chain models [71], neural networks or sequence matching [22].

This dissertation focuses on improving signature-based intrusion detection and response systems. Therefore the next sections will discuss existing signature-based intrusion detection as well as intrusion response. For other ways of classifying IDS be forwarded to [7] and [15]. For an excellent introduction into computer security and intrusion detection be referred to [42].

2.2 Signature-Based Intrusion Detection Systems

Signature-based IDS belong to the class of misuse-based IDS, which means that they perform an exact comparison of some previously defined event with an observed event taken from an event stream. If a 'match' is found, it is assumed that the monitored event contains elements that comprise an attack (see Figure 2.2). An alert is triggered to retain information about the detected event in a log file and to inform the system administrator about possible attacks.

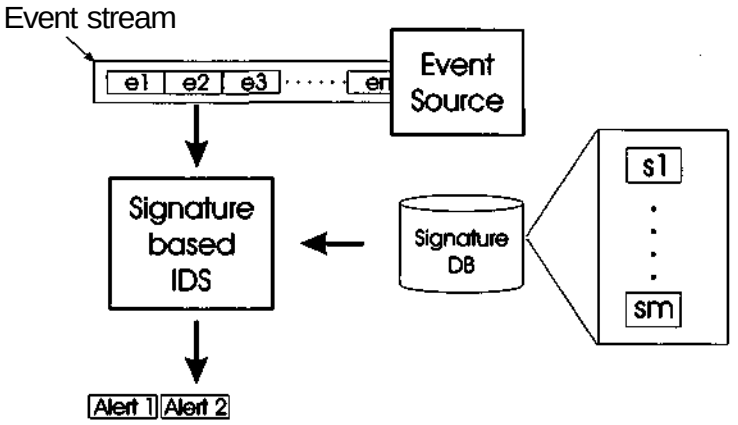


Figure 2.2: Basic Functionality of Signature-Based IDS

Before techniques of current IDS are explained, signature-based intrusion detection is explained formally.

Definition:

A set of signatures 5, consisting of signatures s_i, $1 < i < n$ is related to an event stream E, consisting of individual events e_j, $1 < j < m$ using a function *detect.* The individual signatures *Si* represent the patterns that are assumed to appear in E when attacks are performed. The process of searching the signatures of 5 in F,$F \subset E$, is called 'detection', and is the part where signature-based intrusion detection is performed. For each signature s_i and a set F, the function delivers

$$detect(s_i, F) = \begin{cases} false & \text{if F does not contain } s_i \\ true & \text{if F contains } s_i \end{cases}$$

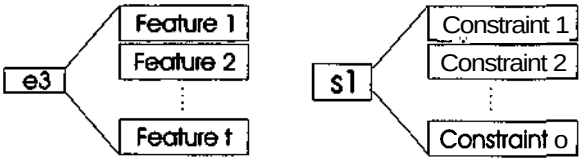


Figure 2.3: Basic Layout of Events and Signatures

The event stream can be collected at various locations within a computer. In general, it can be collected from a host or from a network. In the case of host based ID, one has many options where to collect it exactly; this can range from collecting data from the lowest levels (e.g., examining the content of the memory, collecting OS delivered data, monitoring the behavior of processes) to data collection on very high levels (e.g., inside an application or acquiring data from an application's log files). In the case of network-based ID, only the data packets obtained from the network can be used for data gathering — nevertheless, the individual packets can be reassembled to the corresponding streams and the content of the packets can be decoded and interpreted according to the used protocols.

Depending on the event collection location, different *attributes* are assigned to each event, which are called *features.* Events collected at host level might have features such as *shell-command*, *user-id* or *system-call* to represent relevant event information while events collected from the network level have attributes such as *payload*, *senderIP* or *receiverIP*. A general view on events and signatures is shown in Figure 2.3.

Signatures consist of constraints on an event's attributes — if all the constraints imposed by a signature are fulfilled by the attributes of an event, the event matches the signature. Signatures can, on the one hand, be very simple (e.g., strings). On the other hand, they might also comprise a compound entity that consists of multiple sub-signatures with complex inter-relationships. For both cases it is true that they have to be designed in such a way that they allow expressing constraints upon the events from the event stream. Depending on the type of signature that is used, different methods have to be employed during detection. When attacks can be characterized by specifying a string that has to be in the event, it is sufficient to perform a string search within events; in the case of compound signatures with additional dependencies or relationships, simple string matching is not enough — an efficient search algorithm for fulfilling single signatures and dependencies is required. Additionally, signatures also contain information about what to do in case of a detected attack, which might range from logging the attack to informing the system administrators via text messages or initiation of active countermeasures.

The method of how to choose the set F, which we did not discuss in detail above, varies from system to system and depends on the detection function. As F represents the information which the detection function uses for detecting intrusions the quality of the detection results heavily depends on it. If too little information is given to the detection function, intrusions might not be found; if too much information is used, detection algorithms might not scale to this extent and the detection system will not be able to keep up with the speed that events are generated.

In the simplest case, *F* just contains a *single event* from *E*. Here, the detection system gets an event and tries to match the individual signatures against the content of the event. After the event has been processed, it is abandoned, as it is not necessary to refer to this event again in the future. This method is best suited for detection systems that have to handle events at a very high rate. As each event is investigated separately, the time consuming process of investigating related events is not necessary, resulting in a very high performing system. The clear disadvantage of this method is that only simple intrusions can be detected.

More complex methods for generating *F* use *ranges* of events ($F = \{e_k\}, 1 < a < k < b < n$), where a represents the start of a range and *b* the end of the range. The intent of this method is to be able to express signatures of intrusions that are not detectable by a single event, but can be identified by watching for several events that occur in a small time frame. Therefore, systems using this method usually allow users to define signatures as compound objects. A number of events that occur in a small time frame are analyzed to identify signature matches. For example, a signature might represent two events that are consecutively followed after each other. Each of the individual events is totally legitimate on its own, but the combination of both is dangerous and therefore an indication for an intrusion attempt. During the detection process of such signatures the intrusion detection system has to keep a number of events in an *event-history*, which is needed for finding the individual parts of a signature. The history has a limited size, and as soon as a new event is observed and the size of the history has been exceeded, the oldest events are dropped to make space for new ones and F is updated accordingly. This is similar to the sliding window principle [16] — the events that have entered the sliding window at first are the ones that leave it first (FIFO, first in — first out). So always a 'window' of all observable events is considered for the detection process.

Additional *constraints* among the attributes of events support the specification of dependencies that have to be fulfilled to make a signature match an event, e.g., two events of a network-based signature might have to be caused by the same *senderIP*. As shown in the next sections, the constraints are usually logic expressions that relate the individual attributes of events.

Together with increasing history size the effort to perform signature detection is increased (in many cases even exponentially). As IDS have to work in near realtime, it is prohibitive to use the whole input stream during the detection process. Events are generated at a very high rate and it is impossible to keep all of them in memory. Keeping the history small therefore allows to express more complex signatures than just detection a single event while keeping the search space and the memory consumption relatively low. This clearly has the disadvantage of possibly missing intrusions that are performed slowly, as their individual signs are timely too much scattered to be located in the history.

After this short introduction into signature-based intrusion detection, the relevant signature-based IDS are presented. As the collection location of event streams strongly affects the structure of events and consequently the layout of signatures, systems with a similar event collection location are discussed together.

2.2.1 Host-Based Systems

Host-based IDS collect their event stream — as suggested by their name — at the host level. This means that some sensor is running directly at a host to generate events from a monitored entity. As stated above, events can be collected either at very low levels (e.g., by analyzing the raw content of memory locations) or at higher levels (monitoring the system call behavior of processes) depending on the required information for the detection process.

Virus Scanners

Virus Scanners can be seen as the simplest form of signature-based ID at the host level. Their aim is to prevent a system from being infected with a malicious program by analyzing the program and controlling its execution behavior. Initially, virus scanners performed just a very simple analysis, but soon it was clear that this was not enough, and therefore much research was done to push the detection capabilities of virus scanners. Today, according to [2], we distinguish between the following anti-virus approaches.

- **Scanners** search files for any of a library of known 'signatures' and monitor file sizes.
- **Heuristic Scanners** look for more general signs than specific signatures such as code segments that are common to many viruses and changes of hash values or checksums of files.
- **Activity Traps** stay resident in memory and look for certain patterns of suspicious software behavior (e.g., scanning files).
- **Full Featured Anti-virus Software** applies a mix of the methods explained above, as each of these methods does not have a sufficient detection rate.

In addition to this there are **Emulators,** which are the most recent method for detecting viruses. As sophisticated viruses apply a number of stealth techniques such as polymorphism (here code transformation techniques are applied to change their look every time they replicate) or encryption to avoid being detected, emulation is seen as a way to catch these viruses for which a signature cannot be given. Before a program is executed or a document is opened, the behavior of the program is emulated. This enables the virus detector to look for relevant signatures in the decrypted buffer. Polymorphic viruses can be identified by their common routines. The caveats of emulation are the high cost of emulation and that it is impossible to determine when to stop emulation.

Anti-virus software is activated every time some critical action is performed — if new software is downloaded to a host, if documents that possibly contain macro viruses are opened, etc. Only if the checks reveal that the suspected entity does not contain a virus, the operation is allowed to proceed.

Scanners, Heuristic Scanners and Emulators are very simple regarding the way F is chosen for signature detection, as F consists of only the last recorded event. Activity

Traps and Full Featured Anti-virus Software require the anti-virus software to maintain a history of accessed resources to be able to detect suspicious behavior. Usually this is done by using counters, therefore there is no need to keep a complete event history and F consists only of the last monitored event, as the state is kept external to the processed events. Current virus scanners are designed in such a way that it is possible to keep up with the speed that events are generated, as they only have to make low level checks on these events. Only if program execution emulation is performed it is unclear when to stop, therefore the danger of being unable to monitor events in realtime comes up.

IDA — Intrusion Detection Agent System and DIDS — the Distributed Intrusion Detection System

In [6], a host-based intrusion detection system using mobile agents is proposed. Mobile agents are used for transferring data from several distributed Sensors to a Manager that collects and evaluates it.

IDA looks for traces of events an attacker may have caused and provides its own theory for intrusions. According to the system designers, most remote attackers first try to get some low privileged access to a system and then perform a local attack from there. An attack is divided into four steps.

1. **Search Stage.** During the search stage the attacker scans the host for possible vulnerabilities, e.g., he finds out the versions of the individual services that are running and also checks whether the permissions of configuration files and programs are set in a way that he can misuse it.

2. **Exploit Stage.** In the exploit stage the attacker makes use of information gained during the search stage and breaks into the system (e.g., using exploits of programs such as `finger` or `lpd`).

3. **Mark Stage.** The mark stage is entered as soon as an attacker has broken into the system. Log- or audit files may contain evidence about the break-in.

4. **Masquerade Stage.** During the masquerade stage the attacker 'cleans the system up', so that all evidence about the break-in / illegal access are removed. Usually this is done by tampering log files.

IDA detects intrusions by searching for possible traces that intruders in the mark stage might have caused. Traces, which are referred to as MLSI — Mark Left by Suspected Intruder, found by Sensors are reported to the Manager, that in reaction sends a so-called *tracing agent* to the host from which the MLSI was reported. The tracing agent performs an in-depth analysis of the audit trail by utilizing so called *information gathering agents.* Each information gathering agent looks for different attacks within a user session and reports these results (e.g., where the intruder came from) to the tracing agent. This allows the tracing agent to track the way the user arrived at a system. After having arrived at the

initial host, the tracing agent returns to the Manager and reports the findings there. As agents need to communicate, so-called *Communication Boards* are located at every host. Agents pass messages dedicated for other agents to the communication board that then takes care of the message delivery.

The most interesting elements of IDA are the possibility of tracing users to their origin and the events that comprise an MLSI. To enable tracing of users across a network it is necessary to keep a history of session related information. *F* contains all the relevant information about all *active* sessions that consists of all the events a user has caused between login and logout at a system.

Even more interesting for host-based signature detection are the events that are considered as MLSIs. IDA uses the Berkeley Security Module to obtain relevant events and considers the following items as MLSIs:

1. changes of the effective user id in non-root user sessions,
2. modification of critical files such as /etc/passwd, `/etc/shadow`, `/etc/hosts.equiv` or `/.rhosts`, and
3. startup of /bin/sh, `/bin/cat`, `/bin/make` during network daemon execution.

In summary, IDA offers only a few enhancements compared to virus checkers, as it monitors critical files for modification, watches the points in time when processes are started and can trace intruders back to the origin where they started their attack. Only a few signatures, the MLSIs, are used for the actual detection. This approach can be useful against typical script kiddies that just launch downloaded exploits against a system, as most exploits cause processes such as `/bin/sh` to be launched. On the other hand a sophisticated hacker can modify the attack in such a way that no MLSI is left when the attack is being performed. This is usually done by substituting the commands that generate the MLSI with commands that are equivalent but not suspicious.

As IDA monitors only for the occurrence of very few events, performance of IDA does not seem to be a problem. On the other hand the complicated use of information gathering agents and tracing agents wastes much processing power that could be saved by an integrated approach. Thus, it is not possible for IDA to report intrusions in realtime.

DIDS, the Distributed Intrusion Detection System [69], is very similar with respect to the detection capabilities. Therefore we only explain the differences to IDA.

DIDS does not use mobile agent technology for information gathering. Instead messages are sent to a central entity (the DIDS director). DIDS only looks for failed events, such as failed authentication or failing accesses to resources (to files, network access etc). DIDS is designed to be able to detect attacks that require information from multiple sources, such as so-called *doorknob-rattling* attacks, or password guessing attacks. Here the intruder in general tries a few common account and password combinations on each of a number of computers. After failing a few times, he moves to a new location. The single sensors would only be able to detect a few failed events which are not that suspicious. The director, that accumulates all the information, is able to detect that a much higher number of attempts

have been performed, which is a clear sign of an attacker that tries to hide his break-in. So, it can be detected that within the whole network ten authentication logins failed during a period of two minutes. For identification, DIDS uses so-called *network identifications* that are assigned to each user that enters a network (using telnet, rlogin, ssh etc.). This is passed to the destination host when users move to another host.

In DIDS, *F* essentially contains the events that occur during the fixed monitored time. This is clearly an advantage over IDA, as there is no need for issuing tracing agents.

DIDS has the same detection capabilities as IDA, but only looking for failed events during a fixed time frame will not be able to stop hackers that perform an in depth investigation of the target systems before they attack. Therefore, there will be no notable failed events in the case of trained hacker attacks.

Unfortunately no performance characteristics of DIDS are available. In any case, the DIDS-director is a central entity that has to process all the reported events. If enough hosts are located in a network, too many alerts might be reported so that the DIDS-director is simply overloaded. In such a case, an attacker might hack into a system without being uncovered.

SWATCH

In [26] an approach for performing intrusion detection at the application level was presented. One aim of SWATCH is to ease the monitoring of a large number of log-files that are scattered across several hosts hosts. The second aim is to combat the threat that is caused by attackers that tamper log-files. To tackle these problems, a centralized entity, the simple watchdog collects the individual log messages from the monitored hosts. Each host forwards a copy of each log-file to the centralized entity, where it is stored and can be queried. To keep the load of information that has to be manually processed by the administrator low, so called *filters* help to sort out irrelevant log entries. System administrators therefore only have to analyze the remaining log entries, that possibly represent manifestations of intrusions. SWATCH can be instructed to execute several actions as soon as an interesting log entry is detected, e.g., ring the bell, send an e-mail, execute an arbitrary script etc.

The design goals of SWATCH were to have a system that could be used easily, being able to perform certain actions when an interesting event is detected and reconfigurability at run-time.

The main mechanism of SWATCH is pattern matching. In a configuration file the administrator can specify for which interesting log-entries to query in which log-files using regular expressions. For each expression entry an action has to be specified. SWATCH was implemented in Perl, as it offers its C-like characteristics enhanced with a powerful pattern matcher.

Unfortunately not measurements of the performance are available — it likely that this system will not be able to handle several thousand of signatures, as each regular expression has to be searched for in each incoming log-entry. As SWATCH only performs a search within application log-files, its use is in the domain of detecting configuration errors of

detecting very simple attacks (such as consecutive failed logins). Hackers can definitely find ways of penetrating a system without even generating a log entry.

COAST Project

In [43] [44] a method for performing signature based ID was presented which uses a modified variant of Coloured Petri Nets [33], which they call Coloured Petri Automaton (CPA). The use of CPA is not restricted to the host level — CPA merely can be used at any location to match patterns, because of their generality.

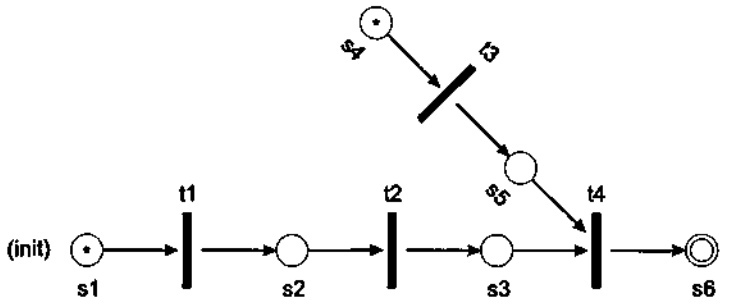
The main goals of this approach have been to be able to find signatures that are temporally distributed over a long period in an effective way. The basic idea is to encode a *scenario,* some legitimate actions which only together form an intrusion, into a coloured petri net [34] that allows efficient search for that scenario. The researchers evaluated Regular Expressions, Deterministic Context Free Grammars, Attribute Grammars and Colored Petri Networks for their purposes and found that only the last one enables ID systems to search for temporally largely distributed patterns, are easily extensible, allow conditional matching and can be easily graphically represented.

Any net in their model requires the specification of more than one initial states, several intermediate states and exactly one final state, all of which are represented as circles. Each circle can be marked with a *token* to represent that this state has been reached. Tokens have attributes (which are strings) that represent the values of variables. The individual states and transitions are connected using arcs, which represent the way transitions can take place. If arcs are directly connected using arcs they are considered to be *e* transitions, which means that they do not require an event to happen. Transitions are represented as thick bars and allow the conditional movement of tokens from state to state, but only if the required conditions are fulfilled. Transitions require an event to happen and the transition's boolean expression to evaluate to true to make the transition fire. When a transition fires, all the attribute values of all incoming tokens are merged and corresponding outgoing tokens are generated.

Matching starts with one token in each initial state. After a token has reached the final state, the pattern represented by the CPA is considered to be matched. For matching it is not required to keep more than one event in an event history, therefore *F* consists only of the last generated event. The pattern matching progress information is kept within the state of the CPA (which consists of the locations of tokens and their attribute values) which is very small compared to the size of the processed events.

The use of CPA allows to express partial orders such as is shown in Figure 2.4. This is done by making use of token attributes and transition conditions. An attribute of tokens can be used to store the timestamps when transitions have been made. At transitions that have more than one input arcs a condition can be set up that checks whether the timestamp of the first token is larger or smaller than the timestamp of the second token. This is remarkable as it allows to model independent signatures that only together form an intrusion. As

Compared to IDA and DIDS the COAST Project provides a more elaborate way of



t1: cp /bin/sh **/usr/spool/mail/root**
t2: **chmod** 4755 **/usr/spool/mail/root**
t3: touch x
t4: mail root < x

Figure 2.4: Sample Scenario of a mail Attack

expressing intrusions. Allowing to specify partial orders provides a way to express much more complex attacks than the ones mentioned in the previous systems. The order in which events have to happen is very important for intrusions to be successful, therefore this method should have a lower false positive rate than the systems before.

COAST Project allows the use of e-transitions and does not restrict the way in which states are connected, which means that the CPA is non-deterministic. But this means that during matching process backtracking has to be performed, as it is not clear in which state the CPA is. The larger the CPA is, the higher will be the cost at runtime for determining the actual state. In the case that a CPA only contains a few states, this is not a problem, but as soon as the size of the CPA grows, the performance will decrease more and more.

The performance of COAST Project unfortunately has not been demonstrated — it is unknown whether it could keep up with the rates events are emerging in usual systems, i.e., whether it can handle these events in realtime. The scenarios that have been shown have all been deterministic finite automatons (i.e., not using e-transitions or having two outgoing transitions from the same node with the same transitions), which means that backtracking is not necessary in these cases. Hence, if more than the few shown scenarios are used it is very likely that the system will not scale, as each monitored event has to be tested against each CPA.

Hackers may have more difficulty in breaking into a COAST Project protected host, as the events that constitute an intrusion are searched for and are also memorized for long times. On the other hand, if the hacker uses attack methods that are not specified as a CPA they cannot be identified.

USTAT

The STAT tool suite [84] is very similar to COAST Project, but also has several important differences. Instead of using CPA, *state-diagrams* are used.

USTAT allows only one initial state in each state-diagram, which has to be determin-

istic. Another fact which differentiates STAT from COAST Project is that STAT allows a more fine grained specification of transitions, which can be consuming, non-consuming or unwinding. In the COAST Project all transitions are consuming, so only one token can perform a transition at a time. In STAT scenarios a transition can be made non-consuming, which means that the incoming tokens of a state are not removed when a transition is performed, which allows concurrent transitions. This causes attacks to be mapped in a more accurate way into the corresponding model, preventing unnecessary false positives.

In contrast to COAST Project the STAT tool suite also provides an extensible language, *STATL*, which allows administrators to specify the attacks that should be identified. The language provides a method for specifying whole scenarios, which consist of states, transitions, assertions and code blocks. Constraints among the individual events can be expressed with assertions of transitions and states and are simple boolean expressions. Therefore partial orders among events can be expressed in a similar way than in COAST Project. For relating the values among attributes, global variables are used.

The scenarios are compiled into a form that allows a rapid evaluation of incoming events and can be inserted at runtime into the system. Timers are used to prevent uncompleted attack scenarios that did not undergo a state change for a long time from using up valuable memory. After a predefined amount of time (which might be weeks) memory consumed by uncompleted attack scenarios is freed. Like in COAST Project *F* just contains the last captured event, as the required information to track evolving scenarios is kept within the state diagrams.

In the following the state transition diagram (as shown in Figure 2.5) of the well known `ftp-write` attack is elucidated. This attack only works if a publicly accessible home-directory is world writable, i.e., everybody can write into this directory. Basically the exploit works by transferring a `.rhost` file into the home-directory. Services such as rlogin use this file to determine whether a remote login is allowed at this machine without the need of authentication. The login process (which is executed by the rlogin facility) reads this file and checks whether the user requesting an rlogin-shell is allowed to get one. If the `.rhost` file is removed after it has been created and before the login process reads this file, there is no intrusion, as the required file is missing.

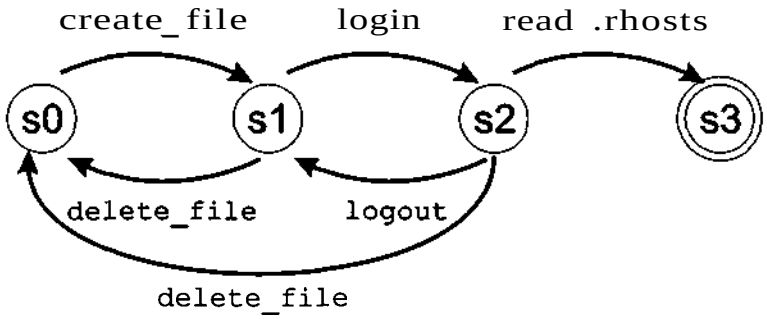


Figure 2.5: USTAT Scenario of the `ftp-write` Attack

By monitoring for this behavior it is possible to

- **detect** the ftp-write attack, and
- **prevent false positives** that would be caused by neglecting operations that remove the file.

Figure 2.5 shows the state transition diagram that models exactly this behavior, for the corresponding text based STATL specification refer to [19].

In short STAT can be seen as the best of the reviewed host-based signature detection engines. The accurate modeling of scenarios can prevent a large number of false positives, similar to COAST Project. In comparison to COAST Project only deterministic state diagrams can be used, which makes an evaluation of incoming events very efficient. The distinction between the three types of transitions (i.e., consuming, non-consuming and unwinding) provides a precise way of specifying scenarios, which can be constructed in a way that minimizes the false positive rate. The high configurability allows security experts to model intrusions very accurately in the intuitive language STATL.

Like every other signature-based system, STAT suffers from the fact that it cannot detect unknown attacks which is its major shortcoming. When an event is detected, it is fed into all scenarios, perhaps performing transitions. This clearly reduces the scalability, as the event processing time is at least linear to the number of instantiated scenarios which can be large.

2.2.2 Network-Based Systems

Instead of monitoring suspicious events at the host level, network based systems monitor network packets for some suspicious activity. Local networks usually operate in such a manner that a network packet is broadcast to all the hosts participating in the network. Therefore, each host actually gets not only its own packets but also the ones directed to the other hosts; filtering is performed at the data link layer (the network interface card) and only the packets that are destined for a certain host are delivered to the network stack. Nearly all network interface cards support a *promiscuous mode* in which all the received data is delivered to the network stack. Therefore, it is not necessary for network-based systems to be deployed on every host — instead, a single host with the NIC's promiscuous mode turned on can monitor traffic of a whole LAN segment. This is a big advantage as a whole network can be monitored *passively*, making network-based systems easy to deploy. In the case of switched Ethernet, where network packets are not broadcast on the communication medium, the switch usually provides a *Tap*-port that allows to monitor all the packets that are passed over the network.

As an analysis of individual packets from stream oriented protocols (such as TCP) cannot give results one can rely on (except in the case of denial-of-service attacks), stream reassembling is performed. This prevents detection systems that only analyze single packets to be mislead, which was demonstrated in [60]. Reassembling and the evaluation of individual packets (e.g., UDP packets) has to be done very carefully. As pointed out in the aforementioned paper, it is possible to craft packets in such a way that network intrusion

Systems and network stacks would disagree about the validity of packets. For hackers it is possible to modify packets in such a way that either the intrusion detection system or the network stack simply reject packets. On the one hand if the intrusion detection system rejects packets that the network stack accepts, the intruder is able to hide the attack packets from the detection system — being able to hack into a system without being noticed. On the other hand, if the attacker sends packets that only the intrusion detection system accepts, the attacker can desynchronize the network intrusion detection system and the network stack.

Network-based intrusion detection works only correctly if the IDS maintains a correct view of the network's state. The attacker can, for example, make the IDS believe that a connection between a server and a client is already closed although in reality it is not. This can be achieved when packets that close an existing connection are sent to a system. If only the NID accepts these, an inconsistent view of the intrusion detection system on the current state has been reached. The IDS will just throw away packets that belong to this connection, allowing the hacker to attack a system undetected.

Network-based signature detection systems are the network level equivalent to host-based signature detection systems. They search for patterns within the events observable at the network level. They look at the packet flow, at the content of packets and analyze whether the specifications of used protocols are met. In the following a comparison of the best network-based signature detection systems is given. Note that network-based systems usually construct F just from the last monitored event.

Snort, Bro and NFR

Snort [66] as well as Bro [58] and NFR [55] [64] are all network-based signature detectors that work on a single host. They are very much the same, therefore we introduce them together. All three systems are divided into four major layers.

1. **Network sniffer.** This layer is responsible for reading out the data that has been captured by the network interface card. All three systems employ libpcap [77] for this task using BPF filters [48]. Using callbacks, the packets are queued and passed to the detection engines.

2. **Event layer.** The event layer decides whether a certain packet should be investigated by the Decision Engine or not, saving computing power by dropping malformed packets that would not be accepted at the target host. In this layer the required stream reassembling is performed.

3. **Decision engine.** Here it is decided whether the caught packet contains an intrusion. If packets do not match any signature, they are dropped in this stage. In NFR, the Event layer and the Decision Engine layer are implemented as a single layer.

4. **Backend engines.** They get as input the packets that passed the Decision Engine and either log and report the results or use the packets as input for additional engines, for example statistical anomaly detectors.

Each of the systems has different ways for defining signatures, each way having its advantages and disadvantages in respect to expressiveness, understandability, maintainability and size.

Snort uses a purely declarative way for specifying signatures, which makes Snort signatures very compact and easy to understand and maintainable. Each Snort rule consists of a single line expressing the required constraints that must be fulfilled to make an event match a signature, as shown in Figure 2.6.

```
alert tcp any any->any 80 (msg:"CGI-tst-nph-cgi"; \
  content:"cgi-bin/nph-test-cgi?";flags:PA;)

alert tcp any any->any 80 (msg:"CGI-test-cgi"; \
  content:"cgi-bin/test-cgi?";flags:PA;)

alert tcp any any->any 80 (msg:"CGI-perl.exe"; \
  content:"cgi-bin/perl.exe?";flags:PA;)

alert tcp any any->any 80 (msg:"CGI-phf"; \
  content:"cgi-bin/phf?";flags:PA;)
```

Figure 2.6: Snort Rules for Detecting CGI Script Misuse

Each of the four rules raises an alert (indicated by the alert statement) when the following constraints are true. The matched packet has to be a TCP packet that has the Acknowledge Flag (A) and the Push Flag (P) set and is sent between arbitrary hosts to the destination port 80 (the Webserver port). The individual content constraints specify which strings have to be present in a matching packet's payload. When the Decision Engine detected a matching packet, an alert with the respective message from the msg section is written to a log file. Snort signatures are not customizable in such a way that state is kept from packet to packet. Snort relies on special plug-ins to realize this functionality, so state is not kept in the core engine.

Bro's scripting language Bro and NFR's scripting language N-Code both have much more complex methods for specifying signatures, as they use full featured C-like scripting languages. Both of them provide the programmer with the possibility of using global variables, which allows the rule creator modeling state across the matching of different packets. Therefore, it is possible to express compound signatures, which contains different sub-signatures, each being found in different packets of the same virtual circuit and in a predefined order. Hence, it is possible to identify intrusions that require several steps, whereas the occurrence of each single step represents a legal behavior. Clearly, this expressivity has its price, as the high level of expressivity and configurability make signatures larger, more complex to handle for the intrusion detection system, hard to maintain and to read for the untrained user. According to [63] more complex N-Code scripts can run to several hundred lines of source code.

The expressivity of Bro and N-code are equal. Nevertheless, there is a big difference between both. Bro scripts are transformed into C data structures to represent the abstract syntax tree of the scripts and an interpreter is used for evaluating scripts. Scripts in N-Code are compiled into an optimized bytecode that operates on a stack based virtual machine. Signatures represented in bytecode are very small and can be efficiently evaluated on packets.

In Bro, NFR and Snort, each signature is considered as a self contained entity and is also treated as such. Therefore each arriving input element has to be matched in a rule-by-rule way. The time these systems [58],[55] and [64] were written the authors did not see many performance problems. Bro has been tested on a moderately loaded (35 Mb/s) FDDI network, while NFR even had not been tested on more than a 10Mb/s network. While creating performance measurements the developers of Bro monitored a 35 Mb/s traffic, where 80% were dropped at the Network- and Event Layer. Only the 7Mb/s (20% of all traffic) of the overall traffic were analyzed in the Decision Engine, and in this experiment they did not notice any packet drops. During another measurement monitoring a every packet of a 25 Mb/s loaded network, they noticed that their filter dropped 364 packets during half an hour. While testing NFR one the 10Mb/s no packet drops have been noticed.

Today, not only the few ($\sim$ 30) signatures are used that were available these days — Snort in version 1.8.7 for example is bundled with 1579 rules, and this database is growing steadily. At this point it should be clear that a mechanism that compares each packet with all signatures is prohibitive. If NFR or Bro would search for all these signatures, the detection within each packet would require the 50-fold time, as the number of rules increases to the 50-fold. This would effect the systems in such a way that they would not be able to keep up with the speed network traffic is arriving, therefore the number of dropped packets would increase largely.

In contrast to Bro and NFR, Snort tries to reduce the number of rules that an input element has to be tested with. Snort uses the addresses and ports for grouping the rules according to these features. Rules that have the same values for the features are put into the same group, which results in the situation that only a part of all rules have to be compared to input elements. Theoretically, Snort's scalability considering the number of rules would be better than the ones of NFR and Bro, because only a fraction of all rules have to be compared to the packet. Unfortunately it turns out during practical measurements that Snort's performance is not sufficient, contrary to what one would expect. Figure 2.7 shows the ratio of the number of analyzed packets vs. the number of packets sent over the network while continually increasing the number of signatures used. These measurements have been made on a heavily loaded Fast Ethernet, Snort has been run on a Pentium III with 550 MHz, equipped with 512 MB of RAM.

The packet matching ratio never reaches 1.0 (which would mean that every packet is analyzed) during this experiment. Initially, when only a few rules are used, approximately 96% of the packets are analyzed, which seems acceptable as the chance for missing a one-packet exploit is only 4%. When the number of rules is increased, the matched packet ratio drops rapidly. When 1579 signatures are used for detection, only 35% of all packets traveling through the network can be analyzed, the remaining 65% are simply dropped.

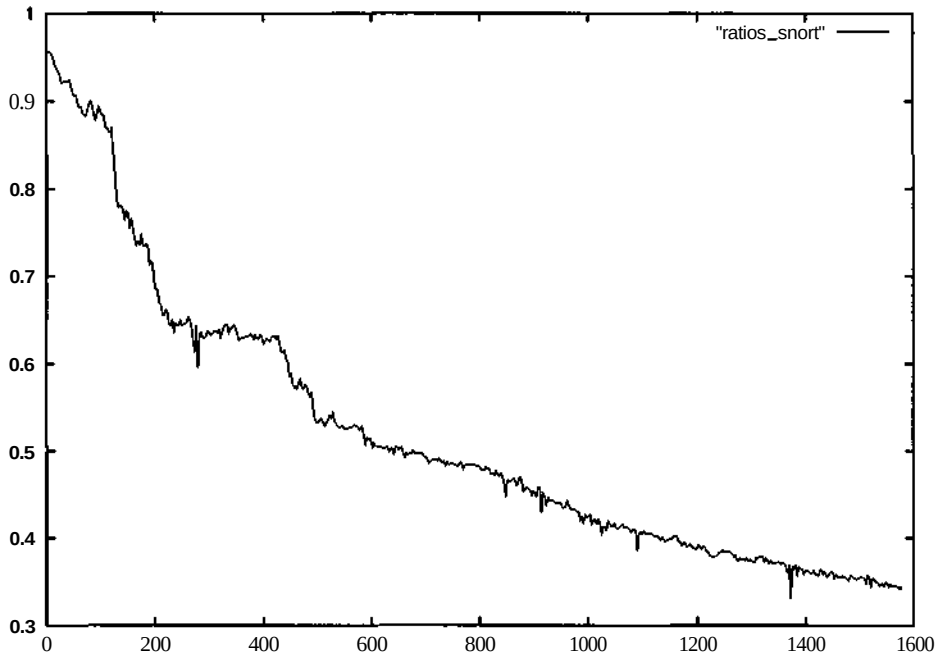


Figure 2.7: Effect of the Number of Rules on the Packet-Matching-Ratio

An in depth analysis of the causes for this bad performance behavior of Snort (although it uses a more efficient approach than for example Bro and NFR) in today's environment will be presented in Chapter 3.

GrIDS

The *Graph-based intrusion detection system for large networks* (GrIDS) [73] has been designed to detect distributed attacks against very large networks by modeling the observable network activity. The focus of GrIDS is on scalability and the ease of integration into these networks (which are large enterprise networks). The data is not forwarded to a central entity (as in DIDS and IDA); instead the network is partitioned into separate nodes that form a hierarchy and each host forwards relevant data to its parent host in the hierarchy. The relevant data does not contain all the individual events that have been observed, as only summaries in form of so-called *activity graphs* are sent upwards. The activity graphs are constructed of nodes and edges, where the nodes represent the hosts and the edges represent the communication behaviors that have been observed. The data for constructing the activity graphs can be obtained from various sources, e.g., network sniffers.

Instead of looking at the content of network connections, the connections themselves are the interesting entities. The main objective of GrIDS is to detect sweeps (such as portscans or doorknob rattling), coordinated attacks or worms, and for doing so the connection patterns are of primary interest.

The individual graphs produced are annotated with attributes, so each connection and each host is assigned attributes that can be used by the graph engine.

For detecting attacks, rule-sets are used that describe how to handle, transform and forward information. For each rule-set separate graphs are maintained, enabling the system a fine grained tracking of individual scenarios. The *precondition* of rule-sets determine which events should be considered to be transformed to graphs and to be added to a graph space (representing partially fulfilled attacks), while the *combining rules* define when and how to merge two subgraphs into one larger graph. Conditions on the attributes of the subgraphs can be formulated to determine when to combine two subgraphs. Formulas are used to describe how to construct the attributes of the combined graph from the attributes of the subgraphs. GrIDS's language provides scripting operations such as manipulation of sets, which are uncommon in IDS scripting languages. Each time subgraphs are combined into a larger graph the resulting graph is evaluated to determine whether it fulfills the *assessments* that describe the signature that is searched for. The assessments also contain conditions that specify when to consider the constructed graph as *relevant*, i.e., when to forward it to the parent host or how to react in the case that a complete signature has been found, e.g., to inform the Network Security Officer. Combining two subgraphs is considered only if both graphs have at least one overlapping edge, and in this case the combination condition is evaluated. If it evaluates to true the combined graph is constructed, while the individual subgraphs are deleted from the graph space (reducing the data size). When a graph is forwarded to a parent host, it is there inserted as a new graph into the engine, perhaps being able to combine this graph with other graphs delivered by other hosts. In order to prevent graphs from growing infinitely, edges are are assigned at timeout, that is reset each time an event representing this edge is detected. If an edge timeout is detected, edges are removed from the graph, possibly dividing it into two subgraphs. This clearly has a big disadvantage, as attacks that are performed very slow cannot be found with this approach (as the relevant edges are removed after the timeout period). If a timeout for a whole graph is exceeded, the whole graph is removed.

As only graphs representing single rule sets are forwarded, no graph engine has to process the whole graph, making the whole system very scalable. Therefore GrIDS offers a scalable aggregation mechanism without putting too much load on single nodes.

An access control system enforces the permissions of users, therefore users can only manage and query the parts that are assigned to them (a collection of nodes from the hierarchy). Users may only add and delete hosts in the hierarchy if they have corresponding permissions to do so.

GrIDS is completely written in Perl and also allows users to import external functions (written in Perl) to be used as correlation methods in assessments and combining rules. Perl is a scripting language that is interpreted during execution, therefore it can be assumed that GrIDS is not dedicated for handling several hundred events per second, as the mechanisms would not be able to cope with this speed. Only the most important attributes (such as timestamps, the used ports etc.) can be handled; an inclusion of a whole event is impossible (e.g., preservation of the used packet payload), as this information would have to be passed to parent nodes, effectively crippling the use of the system.

In Figure 2.8 an example GrIDS scenario is presented, which has been created by the the designers of GrIDS to identify worms. As combining rules as well as assessment rules

are most interesting, only these are shown here.

```
ruleset worm_detector;

 ...

node rule {
  res.node.combine = !empty({new.node.alert, cur.node.alerts}) &&
                     abs(cur.node.time - new.node.time) < 30;
  res.node.alerts  = {cur.node.alerts, new.node.alerts};
  res.node.time    = max(cur.node.time, new.node.time);
}

assessments {
   !empty(res.global.alerts) ==>report_graph(2,"alerts found!");
  res.global.nnodes > 3 ==>
   report_graph(3, res.global.nnodes::" nodes in graph");
  res.global.nedges > 5 ==>
   report_graph(3, res.global.nedges::" edges in graph");

  res.global.nnodes > 7 ==>
   alert(1, "worm detected, >7 nodes in graph from "::res.global.ruleset);

  res.global.nedges > 12 ==>
   alert(1, "worm detected, >12 edges in graph from "::res.global.ruleset);
}
```

Figure 2.8: GrIDS Rule-Set for Detecting Worms

The node rule part describes when and how to combine subgraphs. In this case, for at least one of the two graphs an alert has to have been reported and the last modified timestamps of both graphs have to be within half a minute. For the resulting graph the alert attributes of both subgraphs are merged, the last modified timestamp is set to the newest timestamp of both subgraphs.

The assessments part specifies that if the number of nodes exceeds three or the number of edges exceeds five, then the graph shall be forwarded to parent nodes. If even higher values of these attributes are reached (seven for the number of nodes and twelve for the number of edges), then an alert should be sent to the user interface of GrIDS.

As can be seen easily GrIDS basically searches for the occurrence of situations in which a very high number of active nodes are found. This clearly has the advantage of being able to detect even unknown worms just by their propagation mechanism and the search activity. Nevertheless this method of identifying intrusions is at threat when the communication patterns change. Peer-to-peer software such as file sharing programs open many

connections to other hosts at a time when they are downloading parts of a document from these hosts, therefore it might not be possible to find threshold values that are applicable for the whole network. If the threshold is set too high, intrusions (worms) will be missed — if it is set too low, the false positive rate of the filters will be too high to be acceptable.

In summary GrIDS poses an interesting approach for network intrusion detection by just analyzing attributes of the built activity graphs, as it has the potential of detecting unknown intrusions. The use of activity graphs makes the whole system scalable, although GrIDS is purely made of Perl scripts. On the other hand only a few characteristics can be used for intrusion detection, limiting the overall use. Also, problems arise if intermediate nodes are not working, as events are not forwarded towards the root node, possibly missing intrusions. As the GrIDS prototype only had been installed on the developer's network, not many statements concerning the performance can be made. Because GrIDS is handling events like all other IDS, it does not scale respecting the number of used rules.

NetSTAT

NetSTAT [85] is network-based signature detection system that has been developed for protecting networks that consist of several subnets. The focus of this system is not mainly on detecting single events in a signature stream (although it can do this too), but to detect signatures that are not detectable at a single point in the network. The network-based signature detection systems that have been discussed so far were designed to monitor a whole subnet by setting the network interface card into the promiscuous mode — in the case that several subnets are used this method does not work anymore to monitor all of the traffic. Exactly this point is addressed by NetSTAT. NetSTAT is a tool of the STAT-suite, therefore it allows users specifying network based signatures using state transition scenarios.

NetSTAT basically consists of of a *State Transition Scenario Database,* a *Network Fact Base*, an *Analyzer* and several *Probes.* The *State Transition Scenario Database* contains the individual signatures that are searched for. Each signature describes a scenario that represents some illegal activity and is expressed using STATL. Therefore descriptions contain states, transitions and assertions, as already described in Section 2.2.1. In contrast to USTAT, where all states, transitions and assertion reference just one host, the situation is different in NetSTAT, as different hosts can be referenced. Figure 2.9 shows the state transition diagram that is used for detecting spoofed packets. In the initial state, several declarations and constraints are listed, which include that the attacker and the victim are on different hosts. The transition describes the event in which the attacker sends a UDP datagram from his machine to the victim machine using an IP-address he does not own.

The *Network Fact Base* contains all the relevant information about the network's topology and the running services. The network topology is a description of the constituent components of the network and the way they are interconnected. Basically NetSTAT uses *interfaces, hosts* and *links* as primitive elements, and a whole network is modeled as a graph on the set of interfaces. The interfaces are the nodes of the graph while the hosts and the links represent the edges of the graph. The individual hosts are annotated with

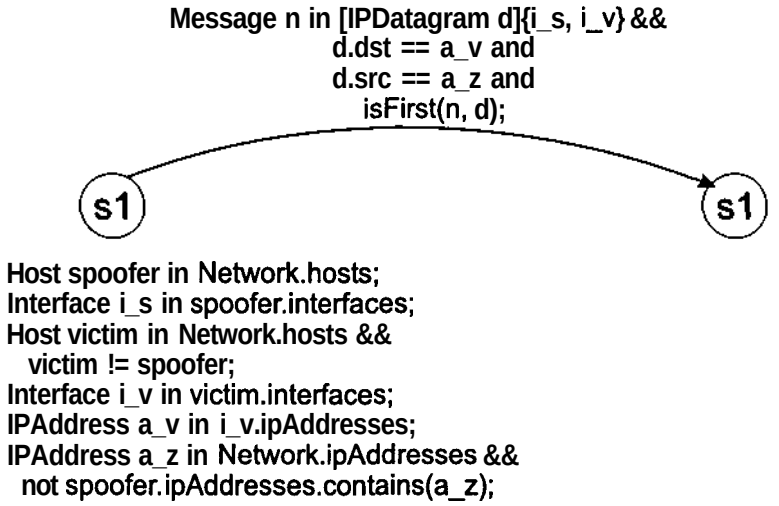


Figure 2.9: State Transition Diagram for Detecting UDP Spoofing Attacks

the services that are running on the respective host (and which are reachable using the interfaces this host is connected to), also stating which protocols these services use and what kind of authentication is utilized.

The *Analyzer is* a component that uses the information of the State Transition Scenario Database and the Network Fact Base to determine the locations in the whole network that have to be monitored for certain events. Therefore NetSTAT works differently from the previously mentioned systems as some offline preprocessing is done, before the actual task of detection can be done. The individual parts of a scenario might only be visible in a certain subnet, and it is the Analyzer's task to identify these locations. On the other hand, not all locations might be usable for searching for a certain event. An event might be totally legal at one location, while the same event might be prohibited and part of an attack scenario, which is caused by the network topology. In the case a scenario is distributed over several nets, it might be necessary for the analyzer to split this scenario and make two or more probes look for parts of the scenarios and configure them to exchange messages that convey state information. The analyzer is also the Network Security Officer's window into the network, as reports about completed scenarios are collected here and a possibility for changing the configuration are given.

Probes are lightweight monitors that are deployed exactly at the locations the Analyzer has determined before. They monitor for the relevant events that have been written into a configuration by the Analyzer. The configuration also contains information about when to pass messages to other probes or to the analyzer to make distributed detection of scenarios possible. Probes basically consist of a network interface that operates in promiscuous mode, a filter that throws away irrelevant events, an inference engine that tracks the completion of scenarios and a decision engine, that has information how to react in the case of a completed scenario.

In short, NetSTAT provides a very convenient tool for performing intrusion detection at

the network level. Once the Network Fact Base and the State Transition Scenario Database are filled by the Network Security Officer, the analyzer automatically determines where to place probes and also their configuration. From the network based systems that were discussed, NetSTAT is the first one that is able to detect scenarios where the individual events have to be collected from several hosts to be able to detect the scenarios. NetSTAT's probes have to filter out events from an event stream — which is done as usual in a linear way by comparing events to signatures iteratively.

SPARTA

SPARTA [40] is an agent based network intrusion detection system that aims at the correlation of information that is distributed across several hosts. SPARTA allows administrators to identify and to relate interesting events at different hosts, searching for intrusion scenarios or just collecting network management information.

Hosts participating in a SPARTA protected network mainly consist of three elements, a *Local Event Generator,* an *Event Storage* and a *Mobile Agent Platform,* as shown in Figure 2.10.

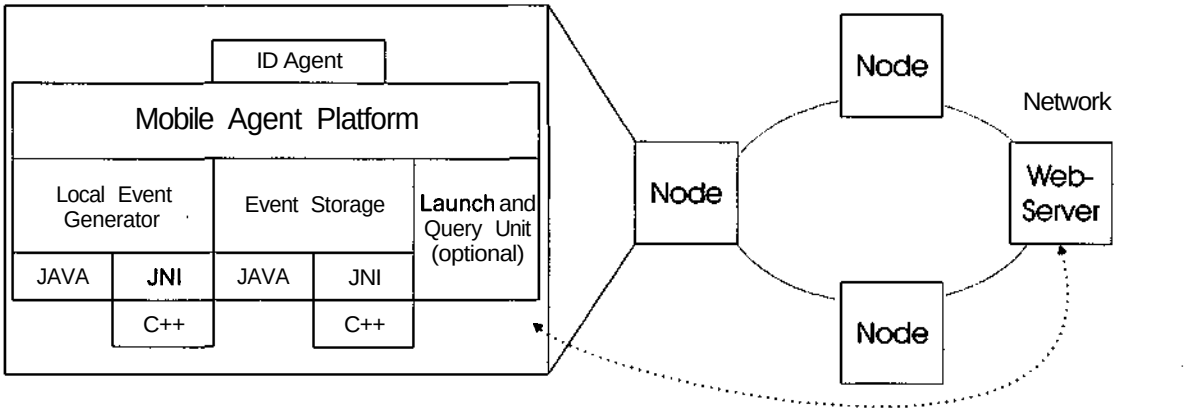


Figure 2.10: SPARTA's Architecture

The Local Event Generator is basically a network sniffer that looks for relevant network packets and forwards the captured packet to the Event Storage. Each event is annotated with attributes, such as id, `timestamp`, the used ports, the used flags, etc.

The Event Storage is a component with the main task to administer events. On the one hand it allows storing events, on the other hand an easy and efficient retrieval of events that fulfill a given specification (as constraints on attributes) is possible. Therefore F contains the events which are currently stored in the individual Event Storages Fi of all t hosts $1 < i < t$ ($F = \cup F_i$). Clearly each Fi has a limited size, therefore events are dropped in *FIFO* order when the reserved buffer has been filled up.

The Mobile Agent Platform (which is based on Gypsy [47], a mobile agent platform developed at the Technical University Vienna using JAVA) allows agents to travel within the network, to collect and to evaluate data. The Mobile Agent Platform can be seen as

an interface between agents and the Event Storage. The Mobile Agent Platform provides several services to agents, such as querying the Event Storage with certain search patterns, providing the host's processing power to the agent and transferring the agent to another host. The Mobile Agent Platform also allows to perform an update of the signatures that are used in the local Event Generator to find relevant events. While agents are transferred over the network, the mobile agent platform ensures that agents are transferred securely and confidentially. The content of agents is encrypted and hosts involved in agent exchange are authenticated. Additionally the mobile agent platform must protect the hosts from malicious agents, therefore the permissions of agents are determined upon whether the agent comes from a trusted host and its authenticity is given.

A so-called *Launch and Query Unit* is placed additionally at the hosts that provide a user interface. The user interface is realized via a Webserver that accepts queries that represent intrusion detection signatures. The query is parsed into an efficient data-structure, checking the syntax and the semantics of the query in parallel. An agent is then assigned to search for all instances of this scenario by passing the created data structure to it, and launching the agent.

For SPARTA, a complex language for describing intrusion scenarios has been developed. The language allows to express complicated, distributed scenarios in a declarative way. Instead of specifying how to detect, only the elements that should be detected are given to SPARTA. The primary goals of the language were to be expressivity on the one hand, but on the other hand to minimize the amount of traffic that is sent over the network. For this reason, the language allows only to express tree-like scenarios, as an algorithm could be found that detects this kind of scenario in an efficient manner.

The correlation mechanism for finding tree-like scenarios does not require central servers, which might be a single point of failure. Instead the agents collect the data at the individual hosts and try to identify the root nodes of the tree-like scenarios. Having found them, the agent can then start to investigate whether the other constraints are fulfilled for a complete scenario match. For this, the agent follows relevant links to other hosts to search there for sub-patterns. All the time they move to another location they take their state (comprised by internal data structures) with them. The agent basically works its way from the root of the scenario to the leaves. An important factor for the correlation mechanism is the use of variables, that allow to relate events that occur at different hosts. If all the evidence for a scenario has been found (which consists of evidence for each event in the scenario), a security policy violation is identified and can be reported using the graphical user interface.

After the Network Security Officer got a report of such a violation, he can gather in depth information about it by either sending additional queries and by evaluating the attributes of the returned data structures.

In contrast to NetSTAT, scenarios can be searched for in an ad-hoc manner, therefore there is no need to analyze the scenarios and to place probes in the network before detection can be performed. Each SPARTA host collects the information it is instructed to, and the query mechanism makes it is easy to determine what has happened.

SPARTA's mobile agent platform is unique compared to other agent platforms, as it has

a full grown public key infrastructure (PKI) which enables to determine the permissions of agents on hosts. An asymmetric encryption scheme is used for guaranteeing the security of agents, and the supported keysizes ($\geq$ 1024 bits) exceed the need of commercial products.

In short SPARTA is an approach that allows users to express complex ad-hoc queries formulated in a custom language. These queries are searched for in a decentralized manner by agents, and the agent platform is secured by a full grown PKI that ensures the security of agents. Instead of forwarding all events to a single node, the information is kept local; when security policy violations are searched for, only the relevant pieces of information are sent over the network, putting very low load on it. SPARTA allows users expressing very complicated scenarios, therefore the run times for agents can be fairly high, as agents have to travel to many hosts. Each time, encryption and decryption as well as authentication has to be performed, making the transfer not as fast as desired. The fact that the capacity of Event Storages is restricted limits the use of SPARTA, as events might emerge with such a speed, that event information is dropped out of Event Storages without having been evaluated by agents. Also the fact that each agent just carries one scenario and the associated information is not optimal — in the case that many scenarios are searched for in parallel, many agents have to travel continuously through a network. Effectively agents are not the optimal solution, as they are only used as containers for information, which can be done with messages more efficiently.

Quicksand

Quicksand [39] can be seen as the successor of SPARTA. Quicksand is (such as SPARTA) a decentralized correlation framework, but tries to overcome several limitations. Quicksand effectively uses the same language as SPARTA, but instead of utilizing agents, simple messages are sent between nodes. As soon as partial scenarios have been detected, a corresponding message is forwarded. In contrast to SPARTA, where the detection algorithm starts at the root node of the scenario, the messages are passed from the leaves towards the root node. Therefore the root node collects the information of different nodes; if the root-node gets all the expected input messages and all constraints are fulfilled, then the scenario is considered to be detected.

Quicksand not only introduces temporal and static constraints, such as SPARTA, but also dynamic ones. Temporal constraints define in which order events have to occur, while static constraints relate the attributes of events from different hosts with each other. Dynamic constraints are introduced by the use of variables in the attack descriptions. If a certain value is bound to a variable, and other parts of an attack description reference this variable in a static constraint, a dynamic constraint representing the static constraint has to be fulfilled. These dynamic constraints are automatically determined from the attack description. A sample of an attack scenario as well the associated dynamic constraints are shown in Figure 2.11.

As soon as all constraints of a node are fulfilled, a message containing the relevant information for the detection of a scenario is forwarded to its successor node.

As the ordering of events at different hosts is crucial for Quicksand, it provides an

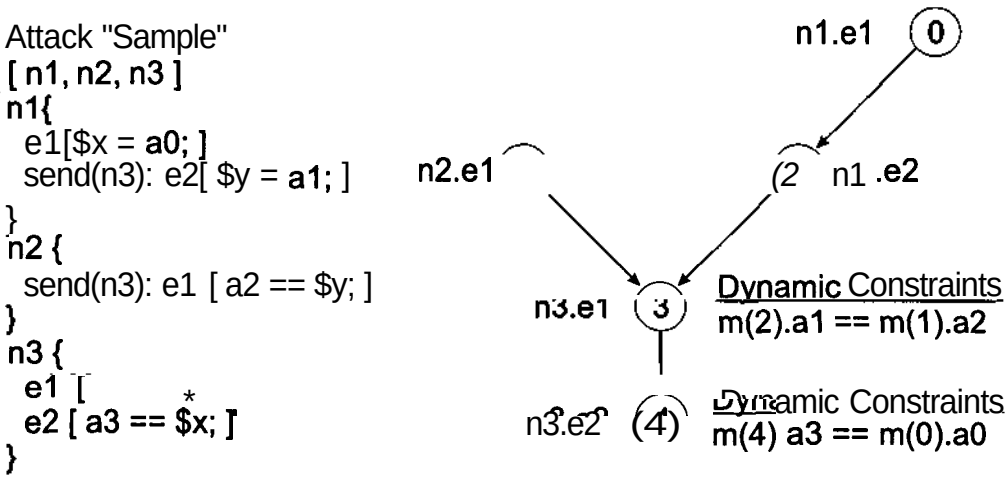


Figure 2.11: Dynamic Constraints in Quicksand

IP-based mechanism for this. Synchronization of all nodes is not necessary, as only the relative ordering of events is required.

In summary one can say that Quicksand is very much like NetSTAT, but it focuses more on the problems that are induced through the relation of distributed events. While NetSTAT is more focused on modeling intrusions as state transitions on a single host, Quicksand assumes that it is rarely the case that two events that are relevant for an intrusion scenario occur on one host — therefore Quicksand also allows to express scenarios that contain partially ordered events, which is similar to COAST Project. Quicksand's architecture is designed in such a way that message passing is the primary mechanism for attack scenario detection, allowing a completely decentralized correlation mechanism with a very low overhead. As Quicksand's objective is being able to correlate in a decentralized way, the usual linear way of comparing events to signatures is used.

2.2.3 Evaluation of Signature-Based Intrusion Detection

After presenting the most important signature-based intrusion detection systems, a summary and a critical view of these is given. Table 2.1 summarizes the individual systems and provides an easy facility for comparing them.

All of the systems have performance problems as soon as more than just a few signatures are used. All of the systems work satisfactorily if only a few number of signatures are used, but as soon as the the number of signatures exceeds a certain limit, the detection engines cannot keep up with this speed. In this case, the intrusion detection system either starts to drop events, or falls back behind the system. If events are dropped, intrusions might be missed, as the determining events are simply not seen by the intrusion detection system. If events are stored and afterwards analyzed by an ID system that cannot keep up with the speed that events are generated, evidence of an intrusion might be present in the event logs, but is not reported yet. The ID system did not complete analyzing former events, therefore

Name	Virus Scanners	IDA	DIDS	SWATCH	COAST	USTAT	Snort	Bro	NFR	GrIDS	NetSTAT	SPARTA	Quicksand
Event Collection													
Host-based	x	x	x	x	x	x							
Network-based							x	x	x	x	x	x	x
Information source													
Memory, Files	x												
Log files		x	x		x								
BSM Audit Data				x		x							
Network Packets							x	x	x	x	x	x	x
Single host	x	x		x	x	x	x	x	x				
Multiple hosts			x							x	x	x	x
centralized			x		x						x		
hierarchical										x			
decentralized												x	x
Event Storage													
none	x			x	x		x	x	x	x	x		x
log file		x	x		x								
FIFO in memory												x	
Signature specification													
single host only	x	x	x	x	x	x	x	x	x				
multiple hosts										x	x	x	x
partially ordering				x								x	x
simple signatures only	x	x	x	x			x						
compound signatures				x		x		x	x	x	x	x	x
declarative signatures	x	x	x	x	x		x					x	x
procedural signatures						x		x	x	x	x		
signature complexity	low	low	low	high	low	high	low	high	high	medium	high	high	high
interpret signatures	x	x	x	x	x	x	x	x		x	x	x	x
compile signatures									x				
Signature detection													
match all signatures	x	x	x	x	x	x		x	x	x	x		
partition signatures							x					x	x
$F =$													
$\{e_k\}$, latest event	x			x	x	x	x	x	x	x	x		x
$\{e_i\}, e_i \pounds$ session		x											
$\cup F_j$												x	
$\{e_k\} \in$ interval			x										
keeps state across detects					x	x		x	x		x		
can detect new attacks										x			
false positive rate	low	low	low	low	low	low	low	low	low	medium	low	low	low
Scalability on nr. of signatures	low	low	low	low	low	low	medium	low	low	low	low	medium	low
Specialization													
tracing users		x	x										
slow attacks				x		x					x		
attack resistance	x							x					
worms, sweeps										x			
Implemented in	C, ASM	C	C	C++	Perl	C	C	C++	C++	Perl	C++	C, Java	C++

Table 2.1: Comparison of different Signature-Based IDS

it cannot identify intrusions that just happened. Such a delay is clearly undesirable, as it gives attackers the chance to compromise a system (and also the ID system) before an ID system can detect them — the realtime requirement of intrusion detection systems is not met anymore. This linear fashion of matching is clearly a problem, as the matching time increases in a linear fashion, too. The *ad-hoc* mechanisms used in systems such as Snort or SPARTA did not prove to increase the scalability respecting the number of signatures, as signatures do not follow the objectives the data structures have been optimized for. Instead of this trivial rule-by-rule based comparison mechanism another approach has to be taken shows a better performance behavior. In Chapter 3 we propose to change the rule-by-rule based approach into an approach that compares rules in parallel.

As can be seen too, signature-based systems cannot be used for detecting unknown intrusions, as they just look for the signatures that are stored in their databases. If a new attack (which is not covered by the used signatures) comes up, signature-based systems cannot detect these attacks. This claim is supported also by the results of the *DARPA Offline Intrusion Detection Evaluation* from 1998, which are shown in Table 2.2, in which the top three IDS have been evaluated. The attack scenarios have been partitioned into *Known Attacks* and *New Attacks.* During a training phase the 'Known Attacks' have been performed, and the IDS creators were given a chance to adapt their systems for being able to detect the Known Attacks. During an evaluation phase the New Attacks, which have not been used during training, as well as the Known Attacks were used. Scans, Denial of Service attacks (DOS), User-to-Root attacks (U2R) as well as Remote-to-Local (R2L) attacks have been evaluated.

Detection Rates	Scans	DOS	U2R	R2L
Known Attacks	92%	80%	64%	80%
New Attacks	88%	22%	66%	8%

Table 2.2: DARPA Offline Intrusion Detection Evaluation Results

As can be seen easily, only 8% of completely new Remote to Local Attacks have been detected, while 80% of the known attacks have been detected. Remote-to-Local attacks are the most feared ones, as they compromise the security of a system without giving other users access to the system. This attack can be performed without any preconditions except that the victim host is running a vulnerable service and is connected to a network. Clearly the inability to detect new attacks is a major shortcoming of misuse-based IDS; exactly this point is addressed by anomaly-based IDS — unfortunately their enhanced detection capability comes with a prohibitive high false positive rate, which makes them unusable in practice. Therefore, all commercial systems utilize signature-based approaches, as they work predictably and have a very low false positive rate.

In the case of upcoming new attacks, the signature databases have to be updated to be able to detect new attacks. Often it is also not very easy to identify the signatures of attacks. Small variations of the attack code cause that the crafted attack signatures are not appropriate for detecting the modified attack anymore. New signatures have to be

created according to the exact image of the modified attack for enabling the ID system to detect the new variants.

This is clearly unacceptable, as the defenders of networks are in disadvantage compared to attackers, as they are always at least one step behind. As stated in the introduction of [37] it should be possible to develop generic/abstract signatures. There it is stated that in theory it should be possible to generate generic signatures, unfortunately only little research has been done in this field.

The advantages of generic signatures are clear — instead of looking for simple patterns in network packets and acting like primitive virus scanners, generic signatures model a whole attack-class. This makes one generic signature equivalent to thousands of individual signatures that cover attack instances of the same class. For this, generic signatures have to include some meta-information about the attack class they describe, such as similarities of all attacks that belong to this attack-class. Once the generic signature could be constructed, a sensor implementation can be made that can determine whether a certain event contains an instance of a generic signature, i.e., an attack. The use of generic signatures also helps to face the performance problems of signature-based IDS, because fewer signatures have to be searched for. The size of the signature database can be considerably reduced if many generic signatures can be found, which makes signature-based IDS much more scalable (assuming that generic sensors can be implemented efficiently).

In Chapter 4 two different generic signatures will be presented, one for detecting buffer overflows and another one for detecting worms.

2.3 Intrusion Response Systems

According to [1] intrusion response is defined as the activities of an organization following intrusion detection, i.e., when an intrusion has been identified. The following tasks are involved in the process of *incident handling*, which are usually executed sequentially:

1. **Initial analysis** is the first phase and starts immediately after an intrusion is detected or suspected and includes roughly estimating the inflicted damage, the source of the intrusion and the methods that have been used.

2. **Containment** means to halt or limit further damage and to restrict the intruder within the compromised systems, so that he cannot penetrate other systems.

3. **Damage assessment** is performed by reconstructing the incident in detail. Here the used vulnerabilities are identified.

4. **Restoration of operations** is afterwards performed. The system is restored to a 'known good state', e.g., by re-installing operating systems or reloading data from backups.

5. **Process or mechanism correction** ensures that the intruder will not be able to abuse the identified vulnerabilities again. This is usually done by patching services, upgrading to the most current software versions or changes in the security policy.

6. **Recovery and summation** is finally performed. A final report is created which includes the above gathered information as well as 'lessons learned'. Recovery and summation also might include legal actions. At this point the incident is closed.

As can be seen, this IR process is typically performed manually, which means, that administrators have to take care of an incident and to perform the above listed actions. This imposes a problem for the operation of computers, as they are usually running all the time, while system administrators can only devote a fraction of their time to incident handling. As the task of intrusion detection has been greatly automated with IDS, it is only a small logical step to create intrusion response systems (IRS), which partially automate the intrusion response process. Not all of the tasks of incident handling can be automated, but at least parts of the initial analysis and the containment can be realized. This has the advantage of being able to at least start the incident handling process and possibly to hinder attackers in intruding a system.

The initial analysis is usually done by the ID system, but also additional analysis steps such as the determination of the attack source can be initiated by the IRS. The containment, which can be seen as a self-defense process of the system, is performed only by the IRS. Here usually vulnerable or compromised system parts are separated from consistent ones by changing the security policy.

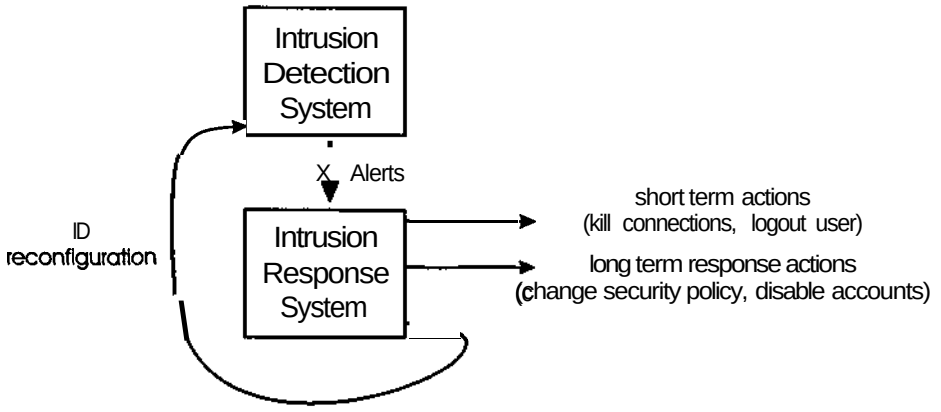


Figure 2.12: Basic Functionality of Intrusion Response Systems

Intrusion response systems (IRS), which are often directly integrated into IDS or at least work close together with them. They take control after signs of an intrusion have been identified and either record the attack or attempt to actively counter it, as shown in Figure 2.12. Although IRS are tightly coupled with IDS and are as important as these in defending against threats, not much research effort has been put into their study; this might also be caused by the fact that intrusion response (IR) is a very difficult problem. In

comparison to ID it is a very young field, and until now researchers have been exploring the mechanisms that can be used to protect networks. In general, the used intrusion response methods can be divided into active and passive ones. While the passive intrusion response methods aim at maximizing the incident information, active intrusion response methods aim at containing the intruder. Table 2.3 gives an overview of the most commonly used intrusion response mechanisms that are used for analysis and containment, taken from [9].

Passive	Active
generate a report	lock user account
generate an alarm	suspend user jobs
enable additional logging	terminate user session
enable remote logging	block IP address
enable additional IDS	shutdown host
create backups	disconnect from the network
	disable the attacked ports or services
	warn the intruder
	trace the connection
	force additional authentication
	employ temporary shadow files
	restrict user activity

Table 2.3: List of common Intrusion Response Mechanisms

2.3.1 Categorization

Current intrusion response systems can be divided into *notification*, *manual response* and *automatic response* systems. The majority of IRS operate as notification systems, which means that they simply display or forward output delivered by the IDS (e.g., incident data) to the system administrator. Usually, urgent notification using alerts is realized via e-mail or text message services over a mobile phone.

Manual IRS allow the administrator to manually launch countermeasures against a detected intrusion by choosing from a predetermined set of response mechanisms. This might allow the administrator to harden the firewall or to change router configurations to disallow malicious traffic. The IRS itself does not decide how to react — instead it provides an easy way for the administrator to quickly implement a consistent security policy. Manual IRS can help to cut off denial-of-service attacks [74] but is also beneficial in the case that the system detects a hacker who has just obtained access to a certain host, as this host can simply be isolated. Such systems support an administrator by offering ready-to-apply reconfiguration mechanisms in order to quickly secure the system (and block further connections from hackers). Nevertheless, a human being has to determine the appropriate methods and to evaluate the effectiveness and the involved risks of countermeasures.

The two categories listed above, notification IRS and manual IRS, are not proactive in countering an intrusion. Even when signs of an intrusion have been reported by the ID system, countermeasures are not triggered automatically and defending the network remains a task for the system administrator. This opens a time window of vulnerability between the point when the intrusion has been detected and the point when the first countermeasure is launched. The size of this time window can range from seconds to hours (e.g., during night times or weekends) and is very important in terms of security. According to [12], the success rate of an intruder rises with the time he can work undisturbed (before the first countermeasures are performed). This interesting study reports that a skilled attacker can perform an intrusion with a 80% success rate if he is given 10 hours time before any response is launched. This number is clearly frightening, as computer systems are not permanently monitored and are often operated over night.

In contrast to the two approaches shown above, automatic intrusion response systems attempt to choose appropriate countermeasures without human intervention. This allows to dramatically reduce the size of the vulnerability window, as IDS monitor for intrusions all the time, and IRS can react within a very short time once an intrusion has been sighted. As no human being is involved in the process of automatic IR, somehow an appropriate response mechanism has to be determined.

2.3.2 Response Mechanism Selection

The way response mechanisms are chosen is very important, as responding to an incident always has some cost [45]. Even in the case that only the logging level is increased, a higher cost can be noticed, as more disk storage and processing power have to be used for logging which could be used for other things. The cost for responding to incidents can be separated basically into three distinct parts. The first part arises from the fact that somehow it has to be determined *how to react* in case of a detected intrusion, which is a nontrivial task. The second part arises from the cost of actually *performing a response.* Simply killing TCP connections is a cheap response compared to tracing an intruder across several networks. The third and last cost comes from the fact that configuration changes and changes in the security policy also can have *negative effects on the usability* of a system and can even lead to a self-denial of service or limited usability. This means that by protecting some resources through attack-countermeasures other resources and services are unavailable and therefore can't help in fulfilling the mission of a network. The first two response costs can be neglected, as the mechanisms causing these costs are very cheap (they are automated). The first two steps of response require just some processing power, sending a few messages across the network and performing some reconfiguration actions. In contrast to this, the third response cost caused by negative effects on the usability of a network can be very high. Services that are usually provided by a network cannot be used anymore because access to them is denied, as a result of a set response.

Assume that in the following e-commerce scenario shown in Figure 2.13 the security infrastructure not only consists of the basic security mechanisms (firewalls, encryption, etc.), but also employs intrusion detection systems that are coupled with automatic response

Systems (which are not shown here for the sake of simplicity, as they can be situated at every host or every network). The mission of the e-commerce site is to sell products 24 hours a day and to give customers feedback about the current delivery status of their orders. For performing these tasks the Webserver has to be reachable from the Internet, the Webserver has to have access to the Database- and the billing-Server, the latter also requires access to an external server that provides a credit card validation service. If any of these constraints are not fulfilled, customers will not be able to use the site's services. If the Webserver is not reachable from the Internet, people cannot to it using their WWW-clients. If the connection between the WWW-server and the Database server is not allowed, the WWW-server cannot fill the generated dynamic pages with the content from the databases. As soon as connections to the billing server or to the external credit card validation services are not possible, customers can only browse the offered goods, but they cannot purchase them. So if any of the just elucidated constraints fail, the intrusion response becomes unusable.

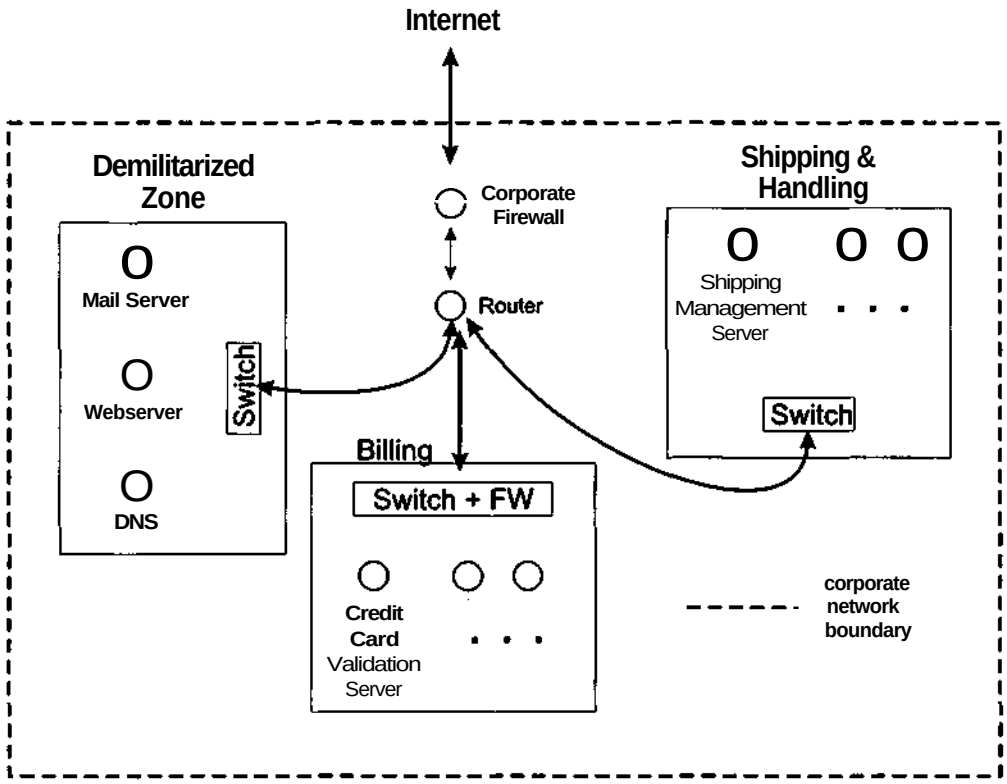


Figure 2.13: Network topology of an e-commerce Site

Currently only two approaches for selecting response mechanisms have been proposed, which are *static mapping tables* and *rules based decision engines.*

In the case that static mapping tables are used for determining responses, each alert is assigned a series of response methods. Every time an alert is found, the corresponding response method is selected and parameterized to create a so-called *response action.* The response action is executed, indifferent of whether the alert is found the first time or has

been found thousand times before, so the IRS always reacts in the same way. Making the response subsystem predictable poses a high risk, as the attacker might be able to find out and abuse the underlying mechanisms. He could attack in such a way that the system performs a kind of self denial-of-service. No measurements are made whether a certain response countering an attack was effective - therefore this approach is completely stateless. As this approach is very simple, most systems follow this approach [9]. Table 2.4 shows a list of intrusion detection systems with automatic intrusion response enabled that use static mapping tables for determining the response.

Name	increase auditing	Sgout User	lock account	terminate user processes	kill connections	disable router interfaces	firewall reconfiguration	trace back to origin	block IP	user specific program
BlackICE [31]	x							x		
CITRA [17]	x	x	x	x	x		x	x	x	x
IDA [6]								x		
IDIOT [43]										x
INSA [78]										x
Intruder Alert [76]	x	x	x							
JiNao [35]	x					x				
NetRanger [54]	x				x				x	
NetSTAT [85]										x
NFR [55]					x		x			x
RealSecure [65]		x	x	x			x			
STAT [84]										x
SWATCH [26]										x
USTAT [29]										x

Table 2.4: IDS with Automated IRS Capabilities

Rule based decision engines work in a more advanced way, as they allow a dynamic selection of the response method, basing on some simple `if-then-else` statements. Co-operating Security Managers (CSM) [87] and Emerald [59] both apply expert systems to perform that task while SecureNetPro [49] provides a scripting language for this. Rule based systems at least allow a more fine grained intrusion response, as response methods can be selected according to conditions on alert attributes. Scripting languages allow the IRS to keep an internal state using variables, allowing to react in a different way when an attack is repeated.

The mentioned mechanisms have the advantage of putting some diversity in the response selection, which decreases the chances of attackers abusing the response system. It is also possible to include the effectivity of launched response actions into the decision process. When a certain response seems to be ineffective against a certain attack, the response mechanism will not be selected when the attack is performed again.

Rule based decision engines also try to mitigate the problem of self denial-of-service, by considering some external information during the decision process. Emerald and CSM

utilize *severity metrics* and *confidence thresholds* during the response decision process. The severity metrics rate all response mechanisms according to their (potential) negative side effects on legitimate network operations. Responses with a high severity level are only allowed when severe attacks are detected with high confidence. The confidence of a detected attack specifies the belief that the detected evidence is an indication of a real intrusion. Confidence thresholds prevent that responses are launched when the confidence is low, as chances are very high that a detected alert is a false positive, and false positives should not be reacted on. If the confidence is low that a certain attack is a 'real' attack, responses to this incident are omitted, otherwise it is performed. A quite similar but different idea is described in [8] where the decision of whether to launch a response is done with the consideration of the expected false positive rate of the underlying ID system. If the ID sensor is expected to have a low false positive rate, the IRS is more likely to invoke severe response actions.

In summary, rule based decision engines can customize the responses to a specific situation much more better that static decision engines. A sample of a decision rule of rule based decision engines is shown in Figure 2.14. Here a response is launched only if the confidence that a buffer overflow attack targeting the apache SSL module has been detected exceeds 90%, the attack has been performed more than three times and the attacker's IP address is not already blocked at the firewall. Otherwise the attacker's IP address is recorded or the alert is simply ignored.

```
alert_apache_ssl_buffer_overflow:
  if ( confidence(alert_apache_ssl_buffer_overflow) > 0.90)

    if (count(list_of_www_buffer_overflow_source_ips,
         alert.sourceIP) > 3) &&
        !ip_blocked_at_host(alert.sourceIP, fw_ip)) {

          block_ip_at_host(alert.sourceIP, fw_ip);
    }
    else {
      insert_ip_into_list(alert.sourceIP,
        list_of_www_buffer_overflow_source_ips);
    }
```

Figure 2.14: A Rule-Based IRS Script

2.3.3 Evaluation of Intrusion Response Systems

Decision tables as well as rule based engines are inherently inflexible mechanisms, as they allow only a static mapping between intrusions and the corresponding response actions (although rule based engines are more fine grained).

In order to provide optimal responses, all possible situations would have to be evaluated and encoded in the rule base. This is clearly feasible only for very small networks with only a few response methods can be set. As soon as the number of governed hosts rises above a certain limit or the already set response actions should be included in the decision process, this approach becomes inefficient. In such a case, the operator would have to perform a manual analysis of all threat scenarios and to evaluate all possible states of the network to determine the rules, leading to a rapid growth of the rule base. If the network is large and the network services become more and more intertwined, hidden dependencies cause the generation of the huge rule set to be more and more cumbersome and error prone. If mistakes occur during the rule creation, wrong response actions are launched during operation, and debugging a huge response rule set is nearly impossible.

The confidence threshold and the severity metrics are too coarse to be able to rely on them. If it is possible to launch a response without a high cost, it should be done. As the severity metrics contain some hard-coded elements, they are not appropriate, because the cost of a response action varies from situation to situation. If only the worst case cost is incorporated into the severity metrics, the network will not be protected as the response cost is too high. If the average cost is used for building the severity metrics, situations might be entered in which the usability of the network is drastically reduced. If response action do not interfere with critical services or when the response does not last longer than a reasonable small amount of time, severity metrics are usable. Unfortunately, responses with long-term effects may seriously hamper regular users from performing their tasks, as desired tasks cannot be performed anymore due to an automatic security policy change.

Current response systems do not take normal operation into account and have no notion of dependencies of services upon each other. Large networks contain many hosts, each running several services which might require other services, making a thorough manual analysis very difficult. The dependencies among the services are usually very complex (as shown in Figure 2.13), involving a number of different protocols. While some services are critical for remote services or users, others may be unavailable for some time without causing problems. Even if a manual analysis is feasible, it would have to be redone every time the configuration of a network changes, which happens regularly.

For current response systems the notion of cost does not exist. Therefore they are unable to choose the best response mechanism with the least cost (with the least negative impact on the network's mission) from several alternative response mechanisms, which would all fit to counter an attack. It is also impossible for rule based systems to determine whether there are some equivalent response methods that provide the same level of security but that have a lower cost.

In summary one can state that IRS are like a two edged blade. On the one hand they give a system a possibility to react upon detected alerts and they are able to perform automated limited incident handling. But on the other hand the rule based approach is not scalable and powerful enough, as the network topology, the already set response actions as well as the resource dependencies within the network are not taken into account. This clearly prevents an IR system from minimizing the response costs. The described severity metrics and confidence thresholds are a first step into the right direction, but they are not

good enough and oversimplified to be of real use.

Therefore it is required to move from this static approach to a more dynamic one, that can automatically handle configuration changes easily without having to manually re-create the large rule set. IRS have to be maintainable in an easy way and have to consider the layout of networks and its resource dependencies to be able to dynamically decide which response method to choose. In Chapter 5 we will introduce a model incorporating the desired characteristics.

2.4 Summary and Conclusions

In this chapter we gave an overview about intrusion detection as well as intrusion response. We presented different signature-based ID systems and showed the common problems of signature-based ID.

1. We showed that all systems have a very limited scalability with respect to the number of used signatures and that

2. signature based systems are unable to detect unknown attacks. All of the presented systems either use ad-hoc approaches for tackling these problems or ignore them at all.

3. The third identified problem concerns intrusion response systems. Finding adequate measure to counter attacks is not an easy task, as imprudent countermeasures can render a whole network worthless. The existing methods are too simple, therefore they don't secure a network in a way in which its usability is kept as high as possible. Therefore the use of automatic intrusion response is a dangerous task if one has to fulfill a mission.

In the following chapters we will tackle these problems.

Chapter 3

Using Decision Trees to Improve Signature-Based Intrusion Detection Performance

A weak man has doubts before a decision, a strong man has them afterwards.

Karl Kraus (1781 - 1832)

In Chapter 2 the basic functionality of signature based IDS was shown. Each monitored event has to be compared to a set of signatures. This process is the most resource intensive one in ID, therefore its overhead has to be kept very low. Signature-based IDS need to have an efficient mechanism for this process to be useful. Scalability respecting the number of used signatures is important, as signatures are coming up on a daily basis. Limited scalability with respect to the number of used signatures clearly decreases the usability of IDS and is a problem — as soon as IDS cannot keep up with the speed events are coming up, either the required real time property of IDS is lost, or events are dropped when the queues (where events are stored before they are processed) are full. Both situations decrease the use of an ID system. In the first case the ID system falls more and more behind the actual system and will not be able to report intrusions as soon as they occur, giving the attacker time to install a rootkit, cover his tracks and shut the intrusion detection system itself down. The second case is not better than the first one, as whole intrusions might be missed, because determining events are dropped. The higher the event drop ratio is, the higher will be the chances to miss intrusions. In Chapter 2 the dependency of the packet drop ratio on the number of applied rules was shown for Snort. The limited scalability caused that more and more packets had been discarded when the number of rules was increased — dramatically decreasing the chances that intrusions or intrusion attempts are recognized.

In this chapter the research results [38] for increasing the scalability of signature-based IDS will be presented. At first, an analysis of the detection complexity of existing (rule-by-rule based) systems is done. We show that in the decision process of existing systems

there is a lot of redundancy, as the same features are compared over and over again. This is followed by sections describing a way of reducing the redundancy vastly using a decision tree based approach, which aims at achieving faster and more scalable systems. The general method for using decision trees in signature-based intrusion detection and the construction of the decision trees are explained. Following is a description of the integration of a prototype implementation into an existing intrusion detection system, Snort. Finally, the approach is evaluated, comparing the performance of the old and the newly gained system.

3.1 Rule-by-Rule Matching

As shown in Chapter 2, all current signature-based systems utilize a linear way of comparing events to signatures (which are also called rules), i.e., they compare incoming events to the signatures/rules in a sequential way. The internal data structure which is used to represent signatures is important, as it determines the way events are matched against signatures, affecting the performance of detection engines.

The simplest technique to determine whether an input element satisfies one of a set of rules is to compare it successively to every rule specification. Such an approach is utilized by STAT [84] or SWATCH (simple watchdog) [26]. Most signature-based IDS store and process signatures in a linear fashion. When an interesting event is captured from the event stream, it is tested against all signatures, possibly fulfilling the constraints of one, triggering an alert.

Consider a STAT scenario that consists of three states, one start state and two terminal states. In addition, consider that a transition connects the start state to each of the two terminal states (yielding a total of two transitions). Every transition represents a rule such that it has associated constraints that determine whether the transition should be taken or not, given a certain input element. In our simple scenario with two transitions leading from the start node to each of the terminal nodes, none, one or both transitions could be taken, depending on the input element. To decide which transitions are made, every input element is compared sequentially to all corresponding constraints. Usually STAT sensors are configured to track several scenarios at once, therefore an input element has to be compared to all constraints of all tracked scenarios in which this element would be applicable. No parallelism is exploited and even when multiple transitions have constraints that are identical or mutual exclusive, no optimization is performed and multiple comparisons are carried out. The same is true for the much simpler SWATCH system. All installed regular expressions (i.e., SWATCH rules) are applied to every log file entry to determine suspicious instances.

This method of internal organization is clearly the easiest one, as each signature is treated and governed as a self-contained entity — but it has the drawback that the *per event overhead* increases linear in the number of used signatures, as with each input element all the rules have to be evaluated. This results in a detection complexity of $O(n)$, where n is the number of used rules.

Some systems attempt to improve this process using *ad-hoc* techniques, but these optimizations are hard-wired into the detection engine and are not flexibly tailored to the set of rules which is actually used. A straightforward optimization approach is to divide the rule set into groups according to some fixed criteria. The idea is that rules that specify a number of identical constraints can be put together into the same group. During detection, the common constraints of a rule group need only to be checked once. When the input element matches these constraints, each rule in the group has to be processed sequentially. When the constraints are not satisfied by the input element, the whole group can be skipped.

This technique is utilized by the original version of Snort, arguably the most deployed signature-based network intrusion detection tool. The developers of Snort utilized this method to reduce the number of comparisons that are needed for evaluating an incoming event. In Snort's internal representation the rules are partitioned according to the used addresses and ports [66]. Snort builds a two-dimensional list structure from the input rules. One list consists of Rule Tree Nodes (RTNs), the other one of Option Tree Nodes (OTNs). The RTNs represent rule groups and store the values of the group's common rule constraints (the source and destination IP addresses and ports in this case). A list of OTNs is attached to each RTN — these lists represent the individual rules of each group and hold the additional constraints that are not checked by the corresponding RTN. The internal organization of the structure is shown in Figure 3.1. When a packet has been caught and should be scanned for signatures, the individual RTNs are searched until matching ones (with the same IP-addresses and ports as in the packet) are found. Then all of the OTNs attached to this RTN have have to be processed, as they are candidates for matching this packet (because the packet has the source- and destination addresses and the required ports for fulfilling the signatures). If none of the OTNs matched, the search is continued at the next RTN. As can be seen, this method has the advantage that whole groups of signatures can be skipped, reducing the overall overhead of processing a single packet.

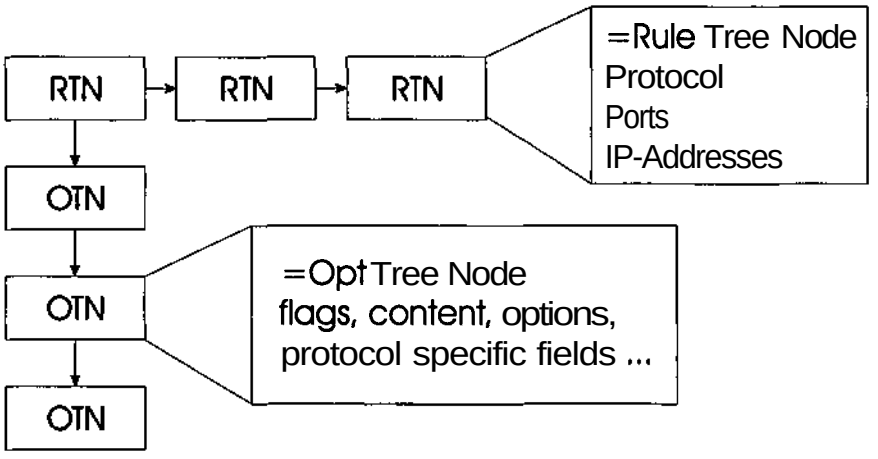


Figure 3.1: Internal Structure for Storing Snort Rules

In theory, the two-dimensional structure could allow the length of the lists, and therefore

the number of required checks, to grow proportional to the square root of the total number of rules. However, the distribution of RTNs and OTNs is very uneven. The 1092 TCP and 82 UDP rules that are shipped with `Snort-1.8.7` and enabled by default are divided into groups as shown below in Table 3.1. The Maximum, Minimum and Average columns show the maximum, the minimum and the average number of rules that are associated with each rule group.

Protocol	# Groups	# Rules	Maximum	Minimum	Average
UDP	31	82	23	1	2.6
TCP	88	1092	728	1	12.4

Table 3.1: Statistics — Snort Data Structures

For UDP, 31 different groups are created from only 82 rules which have only two rules associated to it on average, requiring an input packet to be checked against all of them. For TCP, more than half of the rules (i.e., 728 out of 1092) are in the single group that holds signatures for incoming HTTP traffic. Each legitimate packet sent to a web server needs to be compared to at least 728 rules, lots of them requiring expensive string matching operations.

In the worst case, all signatures are put into the same RTN (when the IP addresses and the ports are not distinguishing), to which a list containing the OTNs of all signatures is attached. Clearly, this is not much better than the method employed by Bro, NFR or STAT, yielding a matching complexity of $O(n)$, where n is the number of used rules.

As can be seen easily, the ad-hoc selection of source and destination addresses as well as ports provides some clustering of the rules, but it is far from optimal. According to our experience, the destination port and address are two discriminating features, while the source port seems to be less important. However, valuable features such as `ICMP` code/type or TCP flags are not used for grouping and are checked sequentially. If in the same RTN two OTNs with the same constraints on a feature (different from the ones used in the RTN) exist, the checks for this feature are redone again at the second feature. This would not be necessary, because the result of the feature analysis on the first OTN would be valid for the second one, too. As some unnecessary processing is redone we can regard this process as containing *redundancy*. This is very undesirable, as the overhead for packet processing is not optimal.

The division of rules into groups with common constraints is also used for packet filters and firewalls. Similar to Snort, the OpenBSD packet filter [27] combines rules with identical address and port parameters into skip-lists, moving on when the test for common constraints fails.

Snort also performs another optimization — in the case that a packet matched an OTN (and therefore a full signature), an alert is generated and the analysis of the packet is considered to be finished — RTNs that have not been analyzed yet are not analyzed anymore. The decision to behave in this way is motivated by the hypothesis that a packet

contains at most one alert. This is actually a controversial 'feature' that has been in the center of much debate ever since. As only a little fraction of input elements match signatures, this premature exit does not speed up the detection process significantly. In addition, a packet that contains a serious attack might trigger another rule first that only reports suspicious, but not necessarily malicious traffic (such as port scans). This is especially dangerous as the order in the rule configuration file is not reflected by the data structures. It is possible that a rule which is specified after another one matches first.

In short it can be stated that although Snort utilizes several technique for avoiding all signatures to be compared with incoming packets, this technique is not optimal. As shown in Figure 2.7 from Chapter 2 the packet filter cannot keep up with the speeds of today's typical networks — therefore this approach has to be improved.

We identified this linear way of comparing events to signatures and the redundancies within the matching process as the main cause for the limited scalability respecting the number of used signatures for current intrusion detection systems. Instead of having a comparison mechanism with a matching time that grows linearly in the number of used rules, a mechanism that can do the same work significantly faster is needed. Redundant comparisons have to be omitted, and therefore every decision step in evaluating an event should allow the ID system to make as much progress as possible in determining whether a signature matches the investigated event. The ultimate goal is to reduce event processing time to speed up the matching process.

3.2 Rule Clustering

The idea of rule clustering allows a signature-based intrusion detection system to reduce the number of comparisons that are necessary to determine rules that are triggered by a certain input data element.

We assume that a signature rule specifies required values for a set of features (or properties) of the input data. Each of these features has a type (e.g., integer, string) and a value domain. There are a fixed number of features $f_1..f_t$ and each rule may define values drawn from the respective value domain for an arbitrary subset of these properties. Whenever an input data element is analyzed, the actual values for all t features can be extracted and compared to the ones specified by the rules. Whenever a data item fulfills all constraints set by a rule, the corresponding signature is considered to *match* it.

A rule defines a constraint for a feature when it requires the feature of the data item to meet a certain specification. Notice that it is neither required for a rule to specify values for all features, nor that the specification is an equality relationship. It is possible, for example, that a signature requires a feature of type integer to be less than a constant or inside a certain interval.

The basic principles have been described in the previous section — possible *ad-hoc* optimizations consider certain features more important or discriminating than others and check on a combination of those first before considering the rest. This technique is used by the original Snort and the OpenBSD packet filter and bases on domain specific knowl-

edge. Nevertheless, the number of required comparisons is still about linear to the number of rules, causing the systems to slow down when the number of rules increases. Unfortunately, novel attacks are discovered nearly on a daily basis and the number of needed signatures is increasing steadily. This problem is exacerbated by the fact that network and processor speeds are also improving, thereby raising the pressure on intrusion detection systems. Much redundancy remained in the matching process of systems that use this ad-hoc grouping mechanism.

We attempt to mitigate the performance problem by changing the comparison mechanism from a rule-to-rule to a feature-to-feature approach. Instead of dealing with each rule individually, all rules are combined in a large set and partitioned (or clustered) based on their specifications for the different features. Compared to Snort, we not only group the rules only one time after a few, hard wired features. Instead we build a complete decision tree, which has multiple levels. By considering a single feature at a time, we partition all rules of a set into subsets. In this clustering process, all rules that specify identical values for this feature are put into the same subset. The clustering process is then performed recursively on all subsets until every subset either contains a single rule or until there are no more features left which could be used for splitting the remaining rules into further subsets. In contrast to the Snort engine, our solution is applicable to different kinds of signature-based systems and not only limited to input from the network. It requires no domain specific feature selection and is capable of parallelizing checks for all features.

3.3 Decision Tree

The subset structure mentioned above can also be represented as a *decision tree*, where the tree's root node represents the set that initially contains all rules while its children are the direct subsets created by partitioning them according to the first feature. Each subset is associated with a node in the tree. When a node contains more than one rule, these rules are subsequently partitioned and the node is labeled with the feature that has been used for this partitioning step. An arrow that leads from a node to its child is annotated with the value of the feature that is specified by all the rules in this child node. Every leaf node of the tree contains only a single rule or a number of rules that cannot be distinguished by any feature. Rules are indistinguishable when they are identical with respect to all the features used for the clustering process.

Consider the following example with four rules and three features. A rule specifies a network packet from a certain source address to a certain destination address and destination port. The source and destination address features have the type IPv4 while the destination port feature is of type short integer.

(#) Source Address——> Destination Address : Destination Port

(1) 192.168.0.1——> 192.168.0.2 : 23
(2) 192.168.0.1——> 192.168.0.3 : 23

(3) 192.168.0.1 ——> 192.168.0.3 : 25
(4) 192.168.0.4 ——> 192.168.0.5 : 80

A possible decision tree is shown in Figure 3.2. In order to create this tree, the rules have been partitioned on the basis of the three features, from left to right, starting with the source address. When the ID system attempts to find the matching rules for an input data item, the detection process commences at the root of the tree. The label of the node determines the next feature that needs to be examined. Depending on the actual value of that feature, the appropriate child node is selected (using the annotations of the arrows leading to all children). As the rule set has been partitioned by the respective feature, it is only necessary to continue detection at a single child node.

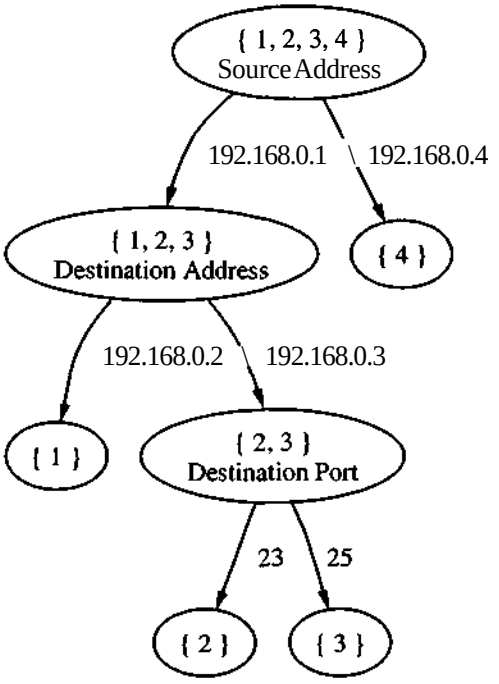


Figure 3.2: Decision Tree

When the detection process eventually terminates at a leaf node, all rules associated with this node are *potential matches.* However, it might still be necessary to check additional features. To be precise, all features that are specified by the potentially matching rules but that have not been previously used by the clustering process to partition any node on the path from the root to this leaf must be evaluated at this point. Consider rule 1 in the leftmost leaf node in Figure 3.2. Both, source address and destination address have been used by the clustering process on the path between this node and the root, but not the destination port. When a packet which has been sent from 192.168.0.1 to 192.168.0.2 is evaluated as input element, the detection process eventually terminates at the leaf node with rule 1. Although this rule becomes a potential match, it is still possible

that the packet was directed to a different port than 23. Therefore, the destination port has to be checked additionally.

At any time when the detection process cannot find a successor node with a specification that matches the actual value of the input element under consideration (i.e., an arrow with a proper annotation), there is no matching rule. This allows the matching process to exit immediately.

3.3.1 Decision Tree Construction

The decision tree is built in a top-down manner. At each non-leaf node, that is for every (sub)set of rules, one has to select a feature that is used for extending the tree (i.e., partitioning the corresponding rules). Obviously, features that are not defined by at least one rule are ignored in this process as a split would simply yield a single successor node with the exactly same set of rules. In addition, all features that have been used previously for partitioning at any node on the path from the node currently under consideration to the root are excluded as well. A split on the basis of such a feature would also result in only a single child node with exactly the same rules. This is because of the partitioning at the predecessor node, which guarantees that only rules that specify identical values for that feature are present at each child node.

The choice of the feature used to split a subset has an important impact on the shape and the depth of the resulting decision tree. As each node on the path from the root to a leaf node accounts for a check that is required for every input element, it is important to minimize the depth of the decision tree. An optimal tree would consist of only two levels — the root node and leaves, each with only a single rule. This would allow the detection process to identify a matching rule by examining only a single feature.

As an example of the impact of feature selection, consider the decision tree of Figure 3.3 which has been built from the same four rules introduced above. By using the destination port as the first selection feature, the resulting tree has a maximum depth of only two and consists of six nodes instead of seven.

In order to create an optimized decision tree, we utilize a variant of ID3 [61, 62], a well-known clustering algorithm applied in machine learning. This algorithm builds a decision tree from a classified set of data items with different features using the notion of information gain. The information gain of an attribute or feature is the expected reduction in entropy (i.e., disorder) caused by partitioning the set using this attribute. The entropy of the partitioned data is calculated by weighting the entropy of each partition by its size relative to the original set. The entropy *ES* of a set *S* of rules is calculated by the following Formula 3.1.

$$E_S = \sum_{i=1}^{S_{max}} -p_i \log_2(p_i) \tag{3.1}$$

where *pi* is the proportion of examples of category i in *S*. S_{max} denotes the total number of different categories. In our case, each rule itself is considered to be a category of its

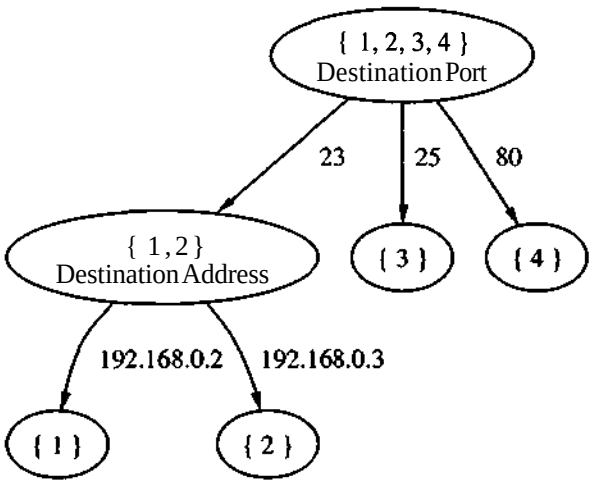


Figure 3.3: Optimized Decision Tree

own, therefore S_{max} is the total number of rules. When S is a set of n rules, pi is equal to $\frac{1}{n}$ and the equation above becomes

$$Es = \sum_{i=1}^{n} -\frac{1}{n} \log_2(\frac{1}{n}) = -\log_2(\frac{1}{n}) = \log_2(n) \tag{3.2}$$

The notion of entropy could be easily extended to incorporate domain specific knowledge. Instead of assigning the same weight to each rule (that is, $\frac{1}{n}$ for each one of the n rules), it is possible to give higher weights to rules that are more likely to trigger. This results in a tree that is optimized towards a certain, expected input.

Given the result about entropy in Formula 3.2 above, the information gain G for a rule set S and a feature F can be derived as shown in Formula 3.3.

$$G_{(S,F)} = E_S - \sum_{v=\mathrm{Val}(F)} \frac{|S_v|}{|S|} E_{S_v} = \log_2(|S|) - \sum_{v=\mathrm{Val}(F)} \frac{|S_v|}{|S|} \log_2(|S_v|) \tag{3.3}$$

In this equation, $\mathrm{Val}(F)$ represents the set of different values of feature F that are specified by rules in S. Variable v iterates over this set. S_v are the subsets of S that contain all rules with an identical specification for feature F. $|S|$ and $|S_v|$ represent the number of elements in the rule sets S and S_v, respectively.

ID3 performs local optimization by choosing the most discriminating feature, i.e., the one with the highest information gain, for the rule sets at each node. Nevertheless, no optimal results are guaranteed as it might be necessary to choose a non-local optimum at some point to achieve the globally best outcome. Unfortunately, creating a minimal decision tree that is consistent with a set of data is NP-hard.

It is obvious that it is not necessary to choose the same order of features when partitioning the branches of the decision tree. It is possible and common that subsets in

different branches of the same decision tree are split by different features, depending on which feature the most discriminating one is.

3.3.2 Non-trivial Feature Definitions

So far, we have not considered the situation of a rule that completely omits the specification of a certain feature or defines multiple values for it (e.g., instead of a single integer, a whole interval is given). As not defining a feature is equivalent to specifying the feature's whole value domain, we only consider the definition of multiple values. Notice that it is sometimes not possible to enumerate the value domain of a feature (such as floating point numbers) explicitly. This can be easily solved by converting the constraints on features into intervals, as shown in Table 3.2. As shown it might be that a single expression is converted into several intervals. `max_feature` represents the maximum value for a certain feature.

original expression	converted ranges
feature = value	[value, value+1[
feature $\leq$ value	[0, value+l[
feature > value	[value+1, max_feature]
feature != value	[0, value[, [value+1, max_feature]
...	...

Table 3.2: Converting an Expression on a Feature to Intervals

When a certain rule specifies multiple values for a property, there can be a potential overlap with a value defined by another rule. As the partitioning process can only put two rules into the same subset when both specify the exact same value for the feature used to split, this poses a problem. The solution is to put both rules into one set and annotate the arrow with the value that the two have in common **and** additionally put the rule which defines multiple values into another set, labeling the arrow leading to that node with the value(s) that only that rules specifies. This process of splitting the range of a feature is shown in Figure 3.4.

Obviously, this basic idea can be extended to multiple rules with many overlapping definitions. The value domain of the feature used for splitting is partitioned into distinct intersections of all the values which are specified by the rules. Then, for each rule, a copy is put into every intersection that is associated with a value defined by that rule. Consider the example rules that have been previously introduced and change the second rule to one that allows an arbitrary destination port as shown below.

(2) `192.168.0.1 ---> 192.168.0.3 :` **any**

The decision tree that results when the destination port feature is used to partition the root node is shown in Figure 3.5. The value domain [0, 2^{16}-1] of destination port has

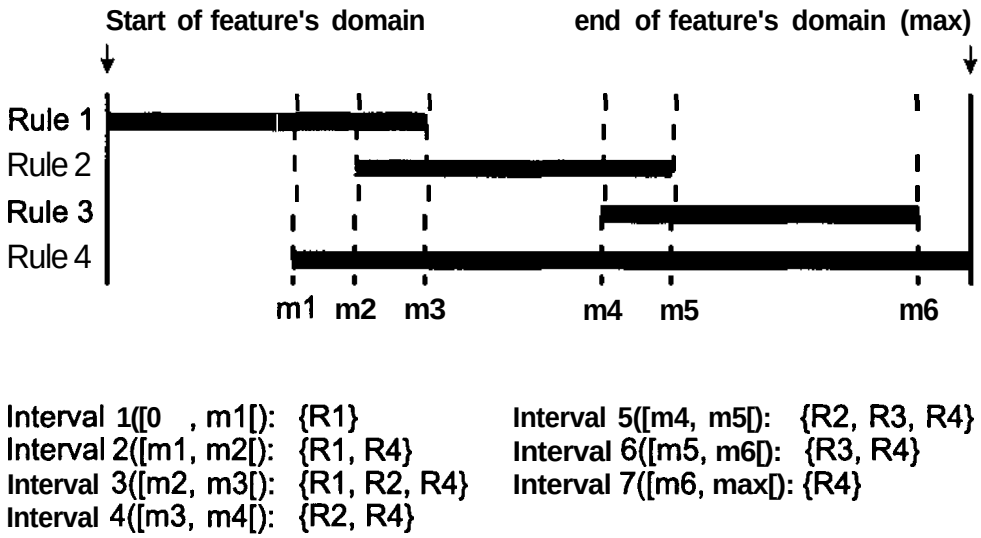


Figure 3.4: Splitting a feature's domain

been divided into the seven intersections represented by the following intervals [0,22], 23, 24, 25, [26,79], 80 and [81, 2^{16}-1]. Rules that define the appropriate values are put into the successor nodes with the corresponding arrow labels. Notice that a packet sent from `192.168.0.1` to `192.168.0.3` and port 25 satisfies the constraints of both rules, number 2 and 3. This fact is reflected by the leaf node in the center of the diagram that holds two rules but cannot be partitioned any further.

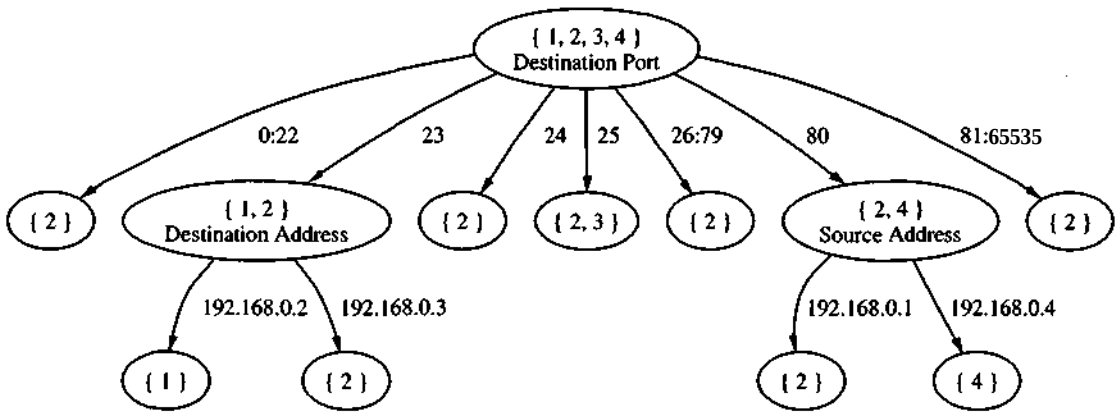


Figure 3.5: Decision Tree with 'any' Rule

The total number of rules in all node's successors does not necessarily need to be equal to the number of rules in the ancestor node (as one might expect when a set is partitioned). This has effects on the size of the decision tree as well as on the function that chooses the optimal feature for tree construction. When many rules need to be processed and each only defines a few of all possible features, the size of the tree can become large. To keep the

size manageable, one can trade execution speed during the detection process for a reduced size of the decision tree. This is achieved by dividing the rule set into several subsets and building separate trees for each set. During detection, every input element has then to be processed by all trees sequentially. For our detection engine implementation, we have used this technique to manage the large number of Snort rules.

3.4 Feature Comparison

This section discusses mechanisms to efficiently handle the processing of an input element at nodes of the decision tree. As mentioned above, each feature has a type and an associated value domain. When building the decision tree or evaluating input elements, features with different names but otherwise similar types and value domains can be treated identically. It is actually possible to reuse functionality for a certain type even when the value domains are different (e.g., 8, 16 or 32 bit variations of the type integer). For our prototype, we have implemented functionality for the types integer, IPv4 address, `bitfield` and string. `Bitfield` is utilized to check for patterns of bits in a fixed length bit array and is needed to handle the flag fields of various network protocol headers.

The basic operation that has to be supported in order to be able to traverse the decision tree is to find the correct successor node when getting an actual value from the input item. This is usually a search procedure among all possible successor values created by the intersection of the values specified by each rule.

Using binary search, it is easy to implement this search with an overhead of $O(\log n)$ for integer, where n is the number of rules. For the IPv4 address and `bitfield` types, the different successor values are stored in a tree with a depth that is bound by the length of the addresses or the bitfields, respectively. This yields a $O(1)$ overhead.

The situation is slightly more complicated for the string type, especially when a data item can potentially contain a nearly arbitrary long string value. When attempting to determine the intersections of the string property specifications of a rule set during the partition process, it is necessary to assume that the input can contain any of all possible combinations of the specified string values. This yields a total of 2^{n-1} different intersections or subsets where n is the number of strings under consideration. This is clearly undesirable. We tackle this problem by requiring that the string type may only be used as the last attribute for splitting when creating the decision tree. In this setup, the nodes that partition a rule set according to a string attribute actually become leaf nodes. It is then possible to determine all matching rules (i.e., all rules which define a string value that is actually contained in the input element) during the detection process without having to enumerate all possible combinations and keep their corresponding nodes in memory.

Systems such as Snort, which compare input elements with a single rule at a time, often use the Boyer-Moore [52] or similar optimized pattern matching algorithms to search for string values in their input data. These functions are suitable to find a single keyword in an input text. But often, the same input element has to be scanned repeatedly because

multiple rules all define different keywords.

As pointed out in [13], Snort's rule set contains clusters of nearly identical signatures that only differ by slightly different keywords with a common, identical prefix. As a result, the matching process generates a number of redundant comparisons that emerge where the Boyer-Moore algorithm is applied multiple times on the same input string trying to find nearly similar keywords. Therefore, they reduce the overhead by using a variation of the Aho-Corasick [3] tree to match on several strings with a common prefix in parallel. Unfortunately, the approach is only suitable when keywords share a common prefix. When creating the decision tree following our approach, it often occurs that several signatures specifying completely differing string properties end up in the same node. Nevertheless, they do not necessarily have anything in common. Instead of invoking the Boyer-Moore algorithm for each string individually, we use a variant of an efficient, parallel string matching implementation introduced by Fisk and Varghese [21]. This algorithm has the advantage that it does not require common prefixes and delivers good performance for medium sized string sets (containing a few up to a few dozens elements).

The Fisk-Varghese approach is an extension of the Boyer-Moore string matching approach, which matches words backwards. In the Boyer-Moore approach a pointer is used for iterating through the string that is searched within (the event string), pointing always to the position where the last character of the searched word is assumed. A skip table is used for determining how many characters can be safely skipped in the event string without missing a word, and the pointer is increased by this number. Figure 3.6 shows this process. The black box marks the position the pointer is currently pointing to. In this example, the character from the position within the analyzed string has been used to determine the number of characters that can be safely skipped (which is four) — the pointer position within the string is adjusted accordingly.

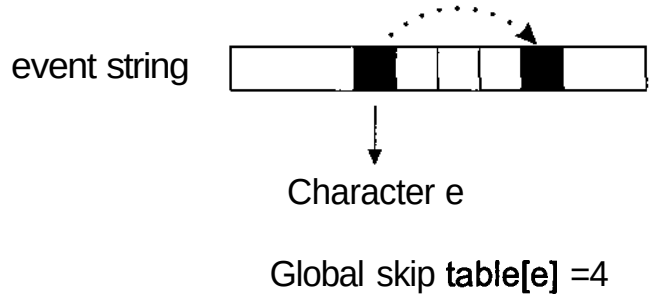


Figure 3.6: Boyer-Moore Search with Skip-Tables

If the skip-table entry of the currently investigated character is zero, the algorithm switches in the exact matching mode. This means, that every character of the searched word is compared to the corresponding character in the event string, starting from the last character (where the pointer is pointing to). If all characters match the searched word, then a match of this string within the event string has been found.

The Fisk-Varghese approach extended the Boyer-Moore approach by searching for several strings in parallel. For this, the skip tables of the individual strings that shall be

searched for are combined into a global one. This global skip table is then used by the pattern matching process to determine the number of characters that can be advanced in the input stream when searching for those string keywords. Individual skip tables for each keyword are built by counting, for each character of the string alphabet, the distance of the last occurrence of this character in the keyword to the end of the keyword. When a character does not appear at all in a keyword, its skip value is set to the length of the keyword. The merge of individual tables into the global skip table is performed by choosing for each character the lowest entry among all individual tables.

The example in Table 3.3 shows the individual skip tables for the strings '`/bin/sh`' and '`.src`' as well as the global skip table after the merge. The global skip table is the only data structure that has to be retained for the detection process.

Character	.	/	b	c	h	i	n	r	s	other characters
'/bin/sh'	7	2	5	7	0	4	3	7	1	7
'.src'	3	4	4	0	4	4	4	1	2	4
Global Skip Table	3	2	4	0	0	4	3	1	1	4

Table 3.3: Building the Global Skip Table From the Individual Word Skip Tables

When the skip value for the current character, which is read from the input stream, yields 0, the detection process cannot advance any further. Now, every keyword has to be matched against the input stream. Fisk and Varghese improved this very expensive process by applying hash tables to this task. Using the minimum length l of all keywords, which can be determined easily when creating the global skip table, a hash value is calculated for the last l characters of each keyword. This value is used as index to insert the keyword into the appropriate slot of the table. Whenever a character that yields a skip value of 0 is detected in the input stream, a hash value is determined for the previous l characters of the stream, including that current one. Now, it is only necessary to perform the exact match between the input stream and those strings which are stored at the corresponding hash table slot, reducing the number of comparisons considerably.

In the Fisk-Varghese approach hash values are generated from the matched content to fight the problem that parts of the input string have to be compared to the individual strings again and again. Only lists of strings with the corresponding hash table entries (which have been previously pre-calculated) have to be compared to the investigated string, reducing the overall overhead. Unfortunately this does not match perfectly with the hundreds of rules that can accumulate in a single hash table entry, when thousands of strings should be searched for. To fight this problem, a selective transformation of the contents of hash table slots into tries, which fan up the strings according to their characters, is performed.

A trie is a hierarchical, tree-like data structure that operates such as a dictionary. The elements stored in the trie are the individual characters of 'words' (which are strings of the individual rules in our case). Each character of a word is stored at a different level in

the trie. The first character of a word is stored at the root node (first level) of the trie together with a pointer to a second-level trie node that stores the continuation of all the words starting with this first character. This mechanism is recursively applied to all trie levels. The number of characters of a word is equal to the levels needed to store it in a trie. A pointer to a leave node that might hold additional information marks the end of a word. If the number of elements in a hash table slot exceeds a certain user definable threshold, all the elements of the hash table slot are inserted backwards into the trie. In the leaves of the trie nodes the strings that have the postfix represented from the root to the leaf of the trie are stored. We limited the depth of the trie, as their use only should reduce the number of strings that have to be matched and as tries consume very much memory.

During the operation, the characters of the event string can be used for identifying potentially matching strings within the trie and then to compare the characters of the searched strings that have not been used during trie building to the event string's characters (because the depth of the tries has been limited). In the leaf node of a trie only a few of the potentially hundreds of strings can be found — only these strings have to be compared character by character with the searched string.

The data-structures and the involved operations during the runtime string matching are depicted in Figure 3.7. At first the global skip tables are used for determining how many characters can be safely skipped. If the skip-value is zero, the hash value is computed from the payload and only the elements in this hash table's slot have to be searched for. If the entries of the hash table slot have been converted to a trie (as shown in our case), the packet's payload is used for navigating through the trie until a leave node is found or the trie does not have corresponding child nodes. If a leave node is found all the elements within this leave node have to be checked using the trivial, character by character based method. The characters that have been used for building the trie do not have to be checked again.

3.5 Implementation

When integrating our data structures and the detection process into Snort, we attempted to keep the changes to the original code as little as possible. This ensures that the modifications can be ported to new versions of Snort easily and enables us to test our components independently of the main program. The two major changes occurred in the parser and in the code that calls the original detection process with its two-dimensional lists.

The parser (i.e., the functions `ParseRule()` and `ParseRuleOptions()`) in `rules.c` had to be adapted to extract the relevant signature information from the rules. Snort translates the checks of properties into function pointers which are later called by the detection process and encapsulates their values in private data areas that have a feature dependent layout. Although possible, it seems undesirable to extract values required by our functions from function pointers and their corresponding private data structures, therefore they are directly gathered during parsing. Nevertheless, the original lists structure is still created and utilized by our code (e.g., for dynamic rule activation) whenever possible.

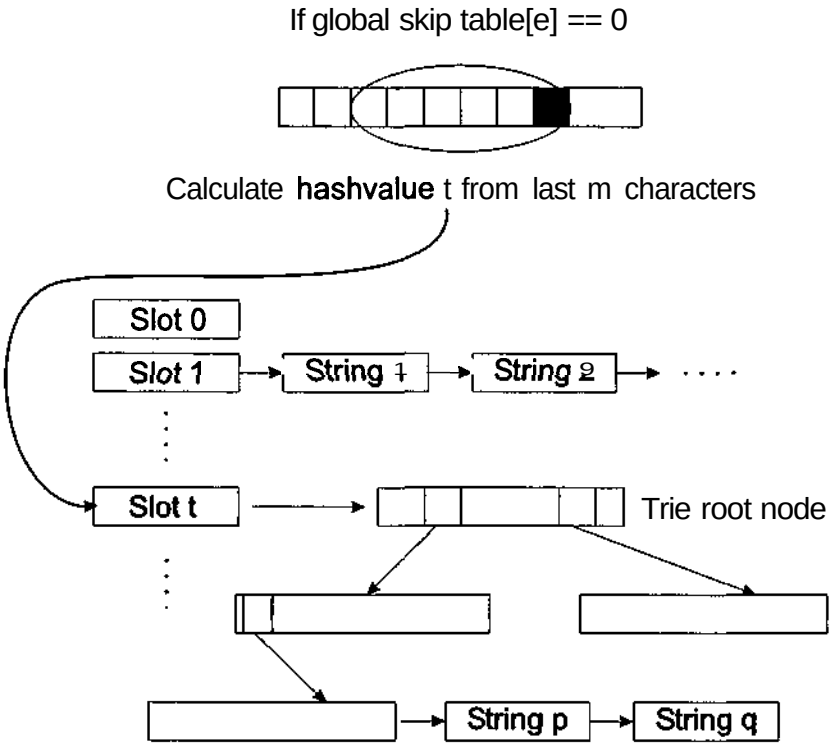


Figure 3.7: Data structures for Efficient String Search

In Figure 3.8 the very basic structure of an OTN is shown. In the original Snort the pointer `opt_func` points to a list of functions that have to be called in order to determine whether the currently analyzed packet matches the constraints in this OTN. The individual functions are called one after the other and get their parameters (which represent the constraints on a certain feature) from a corresponding entry on ds_list. If all of the functions in this list report that the packet matches the packet, then a match has been found. In contrast to the original Snort engine, our modified version already performs the majority of checks that are provided by the function list pointed to by the `opt_func` pointer. So we can remove pointers to the functions that are already realized in our engine from the list pointed to by `opt_func`. Features that are not supported or cannot be made faster by the new detection engine are reused, as depicted in Figure 3.8. Here the original routines for searching a regular expression are reused, while the functions for checking the `ICMP` type and code can be safely removed from this list, as this functionality is provided much more efficiently by the new engine. The newly gained engine has the advantage that it reduced the comparison redundancy vastly and is capable of performing parallel string matches, without making the improved system incompatible to the original one.

The second part of changes affected the detection function (`Detect()`) in `rules.c`. Instead of calling the original processing routine, it redirects to our decision trees. The modified detection procedure calls response and logging functions in a similar way than the old one. However, it is possible that they are called several times for a single packet as

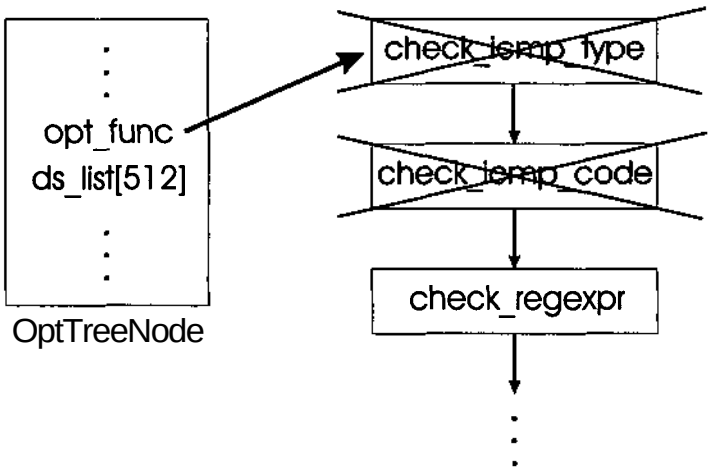


Figure 3.8: Removing Routines of Features that are Covered by the Decision Tree

our engine determines all matching signatures for each input element. When this behavior is undesirable, our module can be put into a mode where only the first match per packet is reported (with the command line switch `-j`). In this mode, our system imitates the original reporting behavior of Snort.

All other changes were only minor modifications of function prototypes to accommodate additional arguments or the addition of variables to data structures such as OptTreeNode. Neither the preprocessing nor the response and logging functionality is affected in any way by our patch. It simply replaces the lists with decision trees. Therefore, it is further on possible to use and write new plug-in modules as desired. In addition, it is also possible to add new features (i.e., to introduce new keywords) to the signature language. Although this seems contradicting at first glance as our decision tree requires the knowledge of these features and their corresponding types, it can be done by excluding these properties from the decision tree and simply checking them afterwards for all signatures that have triggered for a certain packet. This obviously reduces the effectiveness of our approach but allows one to extend Snort and keep the ability of deploying the modified detection engine.

3.6 Experimental Data and Evaluation

This section presents the experimental data that we have obtained by utilizing decision trees to replace the detection engine of Snort. We have implemented patches named Snort NG (next-generation) for `Snort-1.8.6` and `Snort-1.8.7` that can be downloaded from [70]. Our performance results are directly compared to the results obtained with the original Snort 1.8.7.

We performed several experiments. We set up `Snort-1.8.7` and our patched Snort NG with decision trees on a Pentium III with 550 MHz running a SuSE Linux `2.4.18` kernel. Both programs read `tcpdump` log files from disk and attempted to process the data as fast as possible. When performing the measurements, all preprocessors as well as logging

have been disabled, as they might blur the results of the measurements. We wanted to have our results reflect mostly the processing cost of the detection algorithms themselves. We measured the run times of the individual systems on the same log files. For each of these data sets, we performed ten runs and averaged the results. For the experiment, the maximum number of 1579 `Snort-1.8.7` rules that were available at the time of testing have been utilized. Both programs were executed consecutively and did not influence each other while running.

3.6.1 Experimental Data

We performed the following four experiments.

1. We used the 'outside' tcpdump files of the ten days of test data produced by MIT Lincoln Labs for their 1999 DARPA Intrusion Detection Evaluation [46]. These files have different sizes that range from 216 MB to 838 MB and sum up to more than 4 GB of network traffic. For each test set, both systems reported the same alerts. Table 3.4 shows a comparison of all the measured speedups on the traffic when 1579 rules are used. As can be seen easily, the decision trees performed much better than the conventional method for every test case, usually a speedup factor of at minimum of 2.5 can be achieved. The spike at the seventh data set is caused by the fact that nearly 50% of all traffic is SMTP traffic, which decision trees can handle much better than the original approach.

Data set	Snort runtime (s)	Snort-NG runtime (s)	speedup factor
1999, week 4, Monday	131.154558	49.64719	2.64173
1999, week 4, Tuesday	169.429486	63.171321	2.68206
1999, week 4, Wednesday	177.205507	67.429480	2.62801
1999, week 4, Thursday	216.170319	85.179459	2.52600
1999, week 4, Friday	149.813904	60.805611	2.46383
1999, week 5, Monday	304.109436	93.096141	3.26661
1999, week 5, Tuesday	819.446838	108.365601	7.56187
1999, week 5, Wednesday	190.929474	78.072151	2.44555
1999, week 5, Thursday	337.529663	135.979324	2.48221
1999, week 5, Friday	469.698303	188.332626	2.49398

Table 3.4: Comparison of Run Times of Snort and Snort-NG

2. We re-did the first experiment with a modified version of Snort-NG, which had the tries disabled. In this variant hash table slots with many entries have not been converted to tries, causing that many strings have to be compared just for one packet.

For all the data sets listed in Table 3.4 we achieved times which in general were 10% longer.

Besides the overall speedup we also measured the dependency between the number of used rules and the runtime. For this second experiment, we repeatedly measured the run times of the two systems, each time using a different number of rules. The results of this experiment are shown in Figure 3.9.

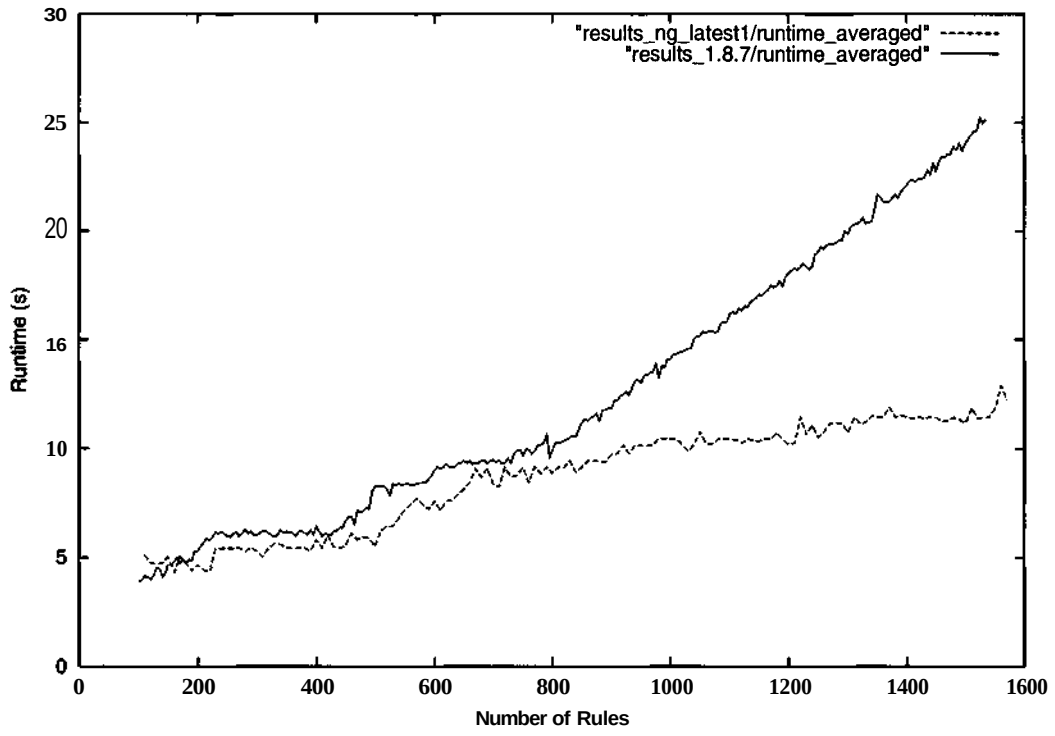


Figure 3.9: Dependency of the Number of Used Rules on the Processing Time

As shown the rule dependency graph of the variant using decision trees are better, as the run times are much lower. Up to 800 rules the run times of both compared systems are nearly identical (although the decision tree based variant is always a little bit better). At 820 rules the run times of the original Snort start to increase more then before, while the run times of the decision tree based Snort are remaining nearly constant. In the interval from 820 to 1570 the run times of the original Snort climb to more than the double of the run-time at 820 rules. Snort-NG's run time on the other hand increases approximately by only 10% in the same interval. From this observed behavior we also expect the curves to continue in the sketched way. Hence the results show, that the approach taken by Snort-NG is much more scalable than the original, as the same files can be processed much quicker by Snort-NG.

3. The third experiment dealt with the worst case scenario of Snort. From the analysis it is clear, that traffic directed to an HTTP server is the one which requires the most

processing power. So we created a big tcpdump file (204488704 bytes) consisting of TCP traffic directed to port 80 only. We then measured the run times with the same setup as just described. The results can be found in Table 3.5.

Data set	Snort runtime (s)	Snort-NG runtime (s)	speedup factor
HTTP requests traffic	2185.4294902221	85.524741	25.55318

Table 3.5: Worst Case Traffic: Comparison of Run Times of Snort and Snort-NG

Snort-NG provides nearly a 26-fold speed increase compared to Snort. The processed file has not been much bigger than the other ones that have been processed in the first experiment. Nevertheless the original Snort required a very long time for processing this file. If Snort was used for monitoring such a traffic, there would have been a catastrophic high packet drop rate (> 90%), voiding the use of Snort. An attacker would have been able to create this kind of traffic to cloak his attacks. The machine running Snort in the victim's network would have been busy analyzing HTTP requests sent by an attacker, letting him to perform his attacks undetected. Snort-NG on the other hand is not affected by this kind of traffic, as the worst case is the usual case. Clustering prevents that at any single node too many elements have to be investigated and limits the number of levels of the trees (which is bound by the number of supported features). This makes the whole system much more predictable and more attack resistant.

4. For the fourth experiment, we connected a client machine (Pentium II with 200 MHz running Linux 2.4) to the computer which was running both IDS (the Pentium III with 550 MHz) using a direct Fast Ethernet connection with 100 Mb/s. In this case, the client replayed the network traffic via `tcpreplay` as fast as possible and both versions of Snort had to capture and process the sent network packets. The goal of this experiment was to study the behavior of the programs under heavy network load on a fully utilized 100 Mb/s link and to measure the number of dropped packets. As could be seen from the previous experiment, the decision trees provided a considerable speedup. The question that should be answered with this tests was how much the packet drop rate can be decreased. Exemplarily we chose one DARPA test data file and measured the effects of the number of used rules on the packet drop rate. Figure 3.10 shows the ratio of analyzed (i.e., successfully processed) packets to the number of transmitted ones. The graphs indicates that both systems lose packets, but the modified version can cope with the load much better. Notice that when only few rules (<200) are used that the original Snort has a better matching behavior than the modified one. As soon as the threshold of 200 rules is exceeded, the modified version behaves better. The number of analyzed packets rapidly decreases, and at 250 rules only 64% of the sent rules are analyzed. The performance of Snort-NG drops gracefully. Using 1579 rules the original version in average analyzed only

82666 of the 233428 packets, yielding a drop rate of 64.587% (analyzing 35.413%). The improved version was capable of analyzing 154705 out of the sent 233428 packets, yielding a drop rate of 33.725% (analyzing 66.275% of the sent traffic). This shows that the decision tree based version really has benefits compared to the old system. The performance increase factor of about 2.5 could not be reached in this experiment. The reason for this is that additional overheads are involved when packets are sniffed directly from the network and not read from disk. Additional decoding has to take place and data has to be transferred from kernel memory to user memory to make it available for user programs, causing a high processing cost.

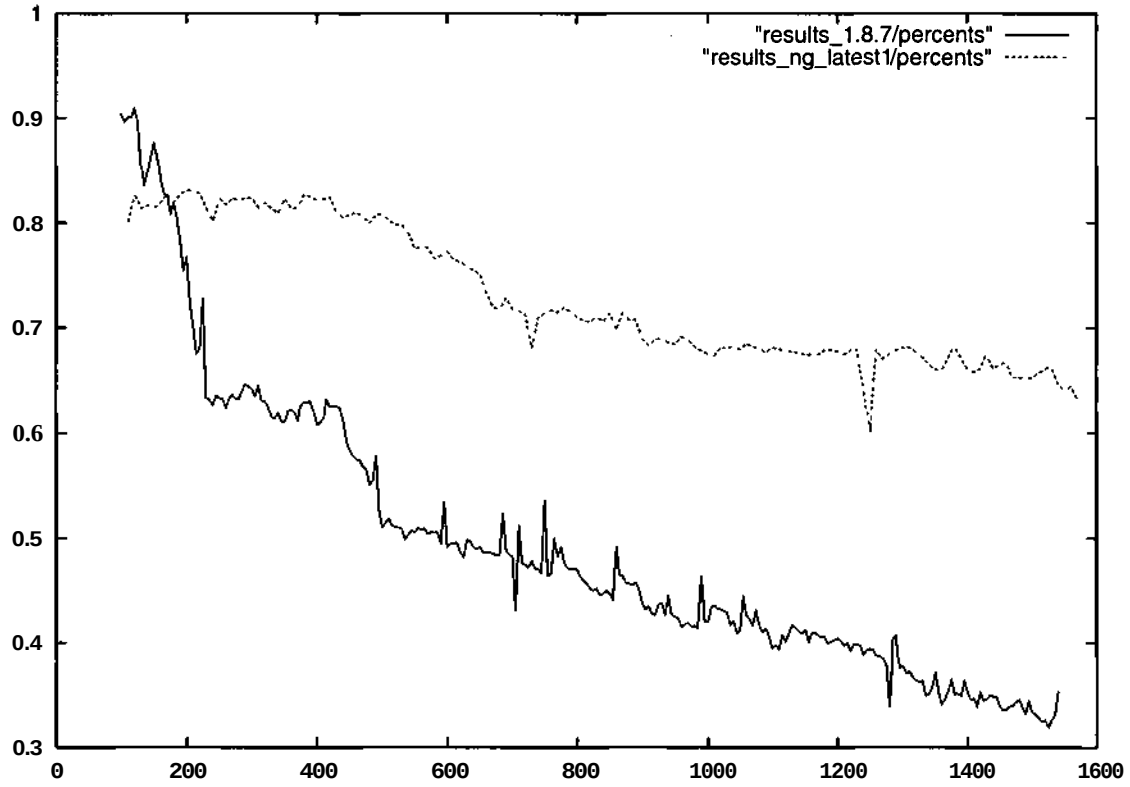


Figure 3.10: Matched-Packet-Ratio Comparison of Snort 1.8.7 and Snort-NG-1.8.7

Building the decision tree requires some time during start up. Depending on the number of rules and the defined features, the tree can contain several tens of thousands of nodes. A few Snort configuration options, such as being able to specify lists of source or destination addresses for certain rules, cause our system to create several internal signature instances from that rule which are later treated independently during the building of the decision tree. When defining a network topology with different subnets and multiple web servers (as needed for the MIT/LL data), the complete rule set used for our evaluation is transformed into 2398 rule instances that need to be processed internally. As a single tree would be too large for this amount of rules, the detection engine splits the rule set for each supported

protocol into two subsets and builds two separate trees. The dependency between the time to build the tree and the number of used rules is shown in Figure 3.11.

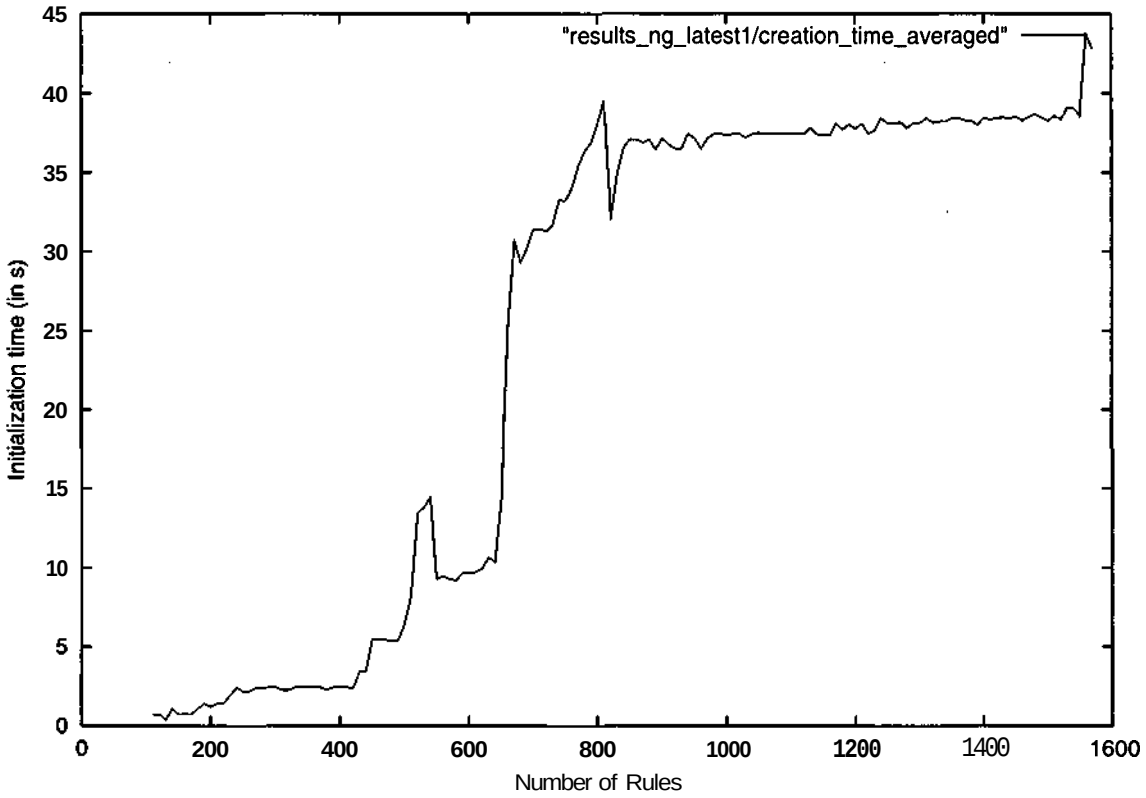


Figure 3.11: Initialization Time

The time to build the tree and all the related data structures (including the case for the maximum number of rules) has never exceeded 45 seconds. At first glance this number seems to be high, as this is outside the range of usual response times. The parsing of the configuration files is done at the same speed as the original Snort does, therefore users get back immediately a confirmation whether the provided configuration file is free of errors. Given this fact and when evaluating that usually intrusion detection systems are started only rarely, as they should be running all the time, this does not seem to be a problem anymore.

Notice the interesting irregularities that Figure 3.11 shows for the modified version of Snort around rule number 550 and 630. The reason for rapid changes in the initialization time is a change in the shape of the decision tree. Given the tree for the previous rules and adding a single additional one, the ID 3 algorithm creates a tree which has the same height but which is much broader. It contains noticeable more nodes (mostly due to copied rule instances with unspecified feature values) and therefore required more time to build. However, additional rules fit well into the resulting tree structure and the detection time does not increase significantly after that as more rules are added (as can be seen in Figure 3.10).

The use of decision trees also increases the memory usage of the program. Figure 3.12 shows the total memory consumption of the patched version of Snort for increasing amounts of rules.

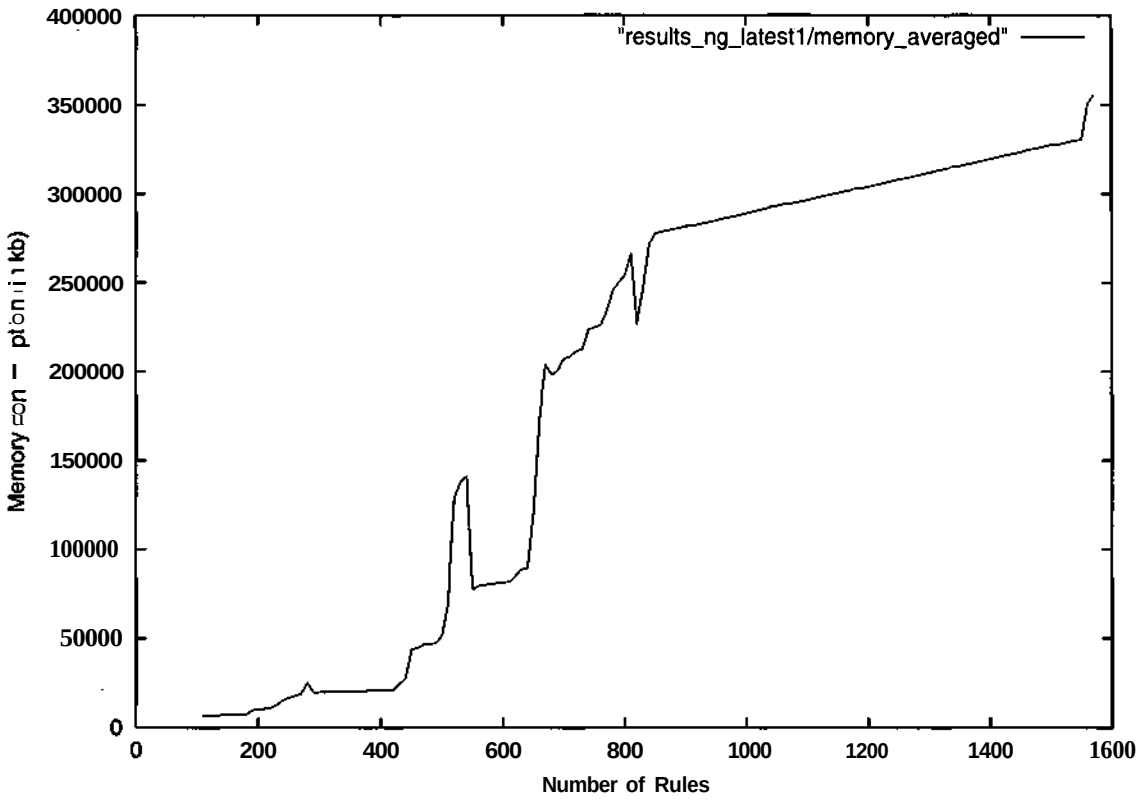


Figure 3.12: Memory Consumption

It indicates that even when the complete set of rules is loaded, the memory demands are reasonable given todays main memory sizes, where whole machines are dedicated for intrusion detection, which are usually equipped with 1 GB of RAM. Notice that similarities in the graphs of the memory usage and the initialization time can be detected. The same spike at rule number 550 exists and near rule 630 the memory consumption suddenly rises very much.

3.6.2 Theoretical Considerations and Evaluation

The number of checks that each input element requires while traversing the decision trees is bound by the number of features, which is independent of the number of rules. This gives a matching complexity of $O(t)$, where t is the number of supported features. The number of supported features is very low compared to the number of rules used, as currently 25 features are supported, but more than 1500 rules are available. However, our system is not capable of checking input data with a constant overhead independent of the rule set size. The additional overhead, which depends on the number of rules, is now associated with the

checks at every node. This is the reason why the run time increases by 10% after rule 800. In the ideal case, the run-time would remain constant once enough rules for building the decision tree have been specified. But as the rules have to be governed by the decision tree, the additional overhead inside the tree is noticeable, although it is very low. In contrast to a system that checks all rules in a linear fashion, the comparison of the value extracted from the input element with a rule specification is no longer a simple operation. In our approach, it is necessary to select the appropriate child node by choosing the arrow which matches the input data value — this usually involves the traversal of a complete feature tree. As the number of rules increases and the number of successor nodes grows, this check becomes more expensive. Nevertheless, the comparison can be made much more efficient than an operation with a cost linear (i.e., $O(n)$) in the number of rules n.

The performance measurements show that our decision tree based approach allowed to decouple the detection time vastly from the number of used rules. Our algorithm has a worst case complexity of $O(t)$, providing scalable intrusion detection concerning the number of used rules, buying processing time by intelligent use of decision trees and by increased memory usage. As long as the tree fits in the main-memory it is possible to achieve a nearly constant matching time, independent of the number of used rules. This enables intrusion detection system to be able to handle more signatures and make them less vulnerable to denial of service attacks in which the attacker overloads the intrusion detection system with its worst case traffic and then performs the attack undetected.

3.7 Summary and Conclusions

In this chapter we presented a way of making signature-based intrusion detection systems much more efficient by applying information theory. We minimized the number of comparisons that are necessary for classifying arriving events and to determine whether they contain intrusions by reducing the redundancy of comparisons. We took an existing intrusion detection system, Snort, and replaced the core detection engine with a decision tree based variant, which allowed an immediate comparison of the old and the new system. The performance measurements showed that the speed of the system greatly improved, and also made the system more predictable, as our approach has a computational complexity of O(t), where q is the number of supported features. We were able to decouple the per event matching time from the number of used rules, which made the whole system predictable and less affected by attacks.

Decision trees, however, are a general solution that can be of benefit to other intrusion detection systems (host- and network-based), packet filters and firewalls as well. Any signature-based ID system needs to categorize incoming events and to determine whether these events contain attack data. Therefore this approach greatly helps in dealing with a large number of rules and can be used by any signature-based intrusion detection system.

Chapter 4

Detecting Unknown Intrusions using Abstract Signatures

In order to improve the mind, we ought less to learn, than to contemplate.

Rene Descartes (1596 - 1650)

Signature-based intrusion detection systems are usually used and maintained as depicted in Figure 4.1. At first, a signature-based intrusion detection system is deployed at some network and it is fed with the current signatures and run.

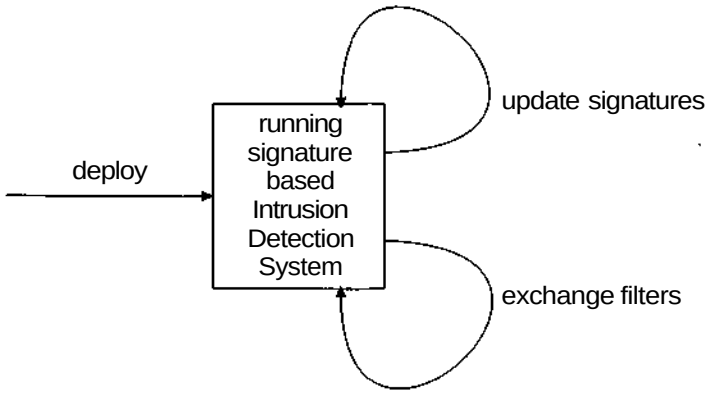


Figure 4.1: Deployment and Maintenance of Signature-Based IDS

When new attacks are detected, they are analyzed by security specialists and transformed into signatures. This is usually done by investigating attack instances over a longer period and then finding out which properties are always the same, such as common strings or byte sequences within the events. This extracted knowledge is transformed to signatures, which exactly describe this identified property. Intrusion detection systems that use a stale signature database base are not able to detect the newly identified attacks, as the signatures of new attacks have not been in the signature database yet. The longer a

signature-based intrusion detection system is not updated, the lower will be the use of the system due to its inability to detect available attacks. This is clearly unacceptable in production environments, as intrusion detection systems should provide additional security, which they fail to in this situation.

Hence it is necessary after some time to update the signature database of the ID system to make it useful again. The rate at which signatures are coming up determines the rate at which signatures have to be updated. As new signatures are coming up on a daily basis, updates of the signature base have to be made at least at the same rate. Intrusion detection sensors are exchanged when better ones are available, but usually the 'patch-and-run' cycle is performed. Large commercial ID system vendors try to keep the time window from the detection of a new signature to the update of the signature databases very low — instead of offering the customers new signatures on the WWW, which would involve a manual update by customers, signatures are directly transferred to the vendor's clients by a push system. In the case that a new attack (e.g., a previously unknown worm) is detected, a signature is manually crafted by security specialists and handed to the push system, which injects it into the running intrusion detection systems of clients.

Although the time window between the availability of new signatures and their use in IDS has been reduced drastically through automation, the situation is not perfect. Signature-based intrusion detection systems are always disadvantaged when compared to a hacker. Hackers usually customize their attack code in such a way that it is not detected by signature-based IDS. Either they write completely new attack code or they modify existing attack code in such a way, that the current signatures (to which hackers have access too) cannot detect the new variant. Therefore the inability of signature-based IDS to detect variants or completely new attacks is a main shortcoming. Anomaly-based IDS tried to fight this problem, as they are able to detect new attacks in principle. Unfortunately the very high false positive rate of anomaly-based IDS prevents their use in typical production environments — for this reason most commercial IDS are signature-based.

The intent of this chapter is to present two different intrusion detection sensors that are capable of even detecting new variants of the same class of attacks. One sensor has been designed for detecting buffer overflow attacks against typical request oriented services such as HTTP, DNS or FTP. A second sensor that has been constructed implements an abstract signature that can detect worms such as Code Red or Melissa. First of all a short introduction into the approaches that have been made for raising the expressivity of signatures is presented. Followed by this an analysis of the two tackled problems is given, followed by a description and an evaluation of both systems.

4.1 Elevating the Expressiveness of Signatures

One approach to fight the problem of constantly having to update the signature database is to make signatures as expressive as possible. Consider a CGI attack, where vulnerable CGI scripts are exploited by making use of the shell's expansion feature. One way of writing signatures that detect CGI attacks is by using the names of the individual vulnerable CGI

scripts as keywords that should be searched for. As more and more scripts are identified as vulnerable, one might want to have a possibility of specifying a more generic way of detecting CGI attacks instead of listing each vulnerable script. An easy solution would be to take the common prefix `'/cgi-bin/'` of CGI scripts as a signature. Definitely this would be a way of detecting all CGI attacks, but on the other hand the sensor operated with this signature would have a squealing high false positive rate, as each CGI access is identified as a CGI attack. This would clearly make the sensor useless.

The idea of generalizing attacks is not new to the intrusion detection community. But soon it has become clear that writing more general signatures that have a low false positive rate is a hard task (as is the same case for anomaly-based IDS). According to [37] only little research has been done in finding some. The aspects of existing IDS that provide some generalization is shown.

In [18] an approach for handling generic signatures was presented using variables. In their special case they wanted to be able to express attacks for which no actual instance could be given. Several operations are committed on the same resource, but under different circumstances (different users access the same files etc.) and always in the same way. So the only similarity is that the same resource is accessed and modified in the same order, which they modeled with their own language using variables. Bro, NFR and SPARTA can do the same, as they all provide a language that allow using variables. In contrast to Bro's and NFR's procedural language, which is specialized on relating consecutive packets from the same stream, SPARTA's declarative language is specialized on connection patterns between different hosts.

NetSTAT automatically determines locations where to place probes by analyzing the scenarios that should be detected and the given network topology. Therefore the scenarios of NetSTAT are generic to a certain extent, as the locations where to place sensors are not fixed.

In the work of [57] generic signatures have been specified on a higher layer and always involved compound signatures (which consist of several sub-signatures). Instead of enumerating the exact sub-signatures for describing attacks, the attack is defined on generalized entities, which they call System View of an attack. A System View can be best compared to a class of an attack. An example showing this kind of specification is shown in Figure 4.2. Here the individual attacks such as *Teardrop Attack*, *Land Attack* and *SYN-Flooding attack* are instances of a class TCPDOSAttack, as they all can be used to perform a denial-of-service attack against a TCP based machine. Instead of defining an attack on the individual attacks such as Land Attack, the attack is defined on the class TCPDOSAttack. To model for example the well known Mitnick attack (which was the example to show their approach), the generic signature is defined on two machines that have established some level of trust and are communicating. When a TCPDOSAttack is detected against one of the machines, the Mitnick attack is detected. In this scenario the TCPDOSAttack shuns one of the communication partners and allows the attacker taking over the attacked communication channel.

This approach using generic signatures has an advantage compared with the previously described approaches, as existing signatures do not have to be modified if new attacks of

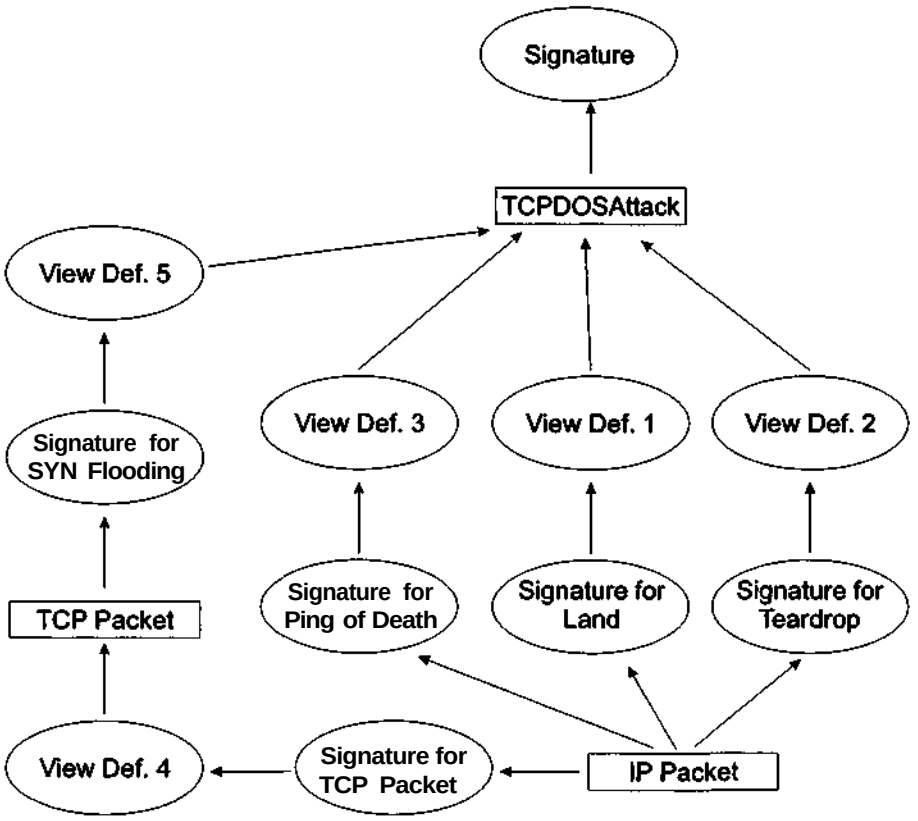


Figure 4.2: Making Signatures Generic Through System Views

the same class come up. If for example new denial-of-service attacks against a TCP stack are discovered, these attacks just have to be added to the class TCPDOSAttack. As the signature has been kept generic, it remains valid all the time, even if new attacks are added to the classes of which the signature is built on. This approach apparently eases rule writing, as it is not necessary to encode the same scenario for different attacks.

From the point of being able to detect unknown attacks, which is one of the main problems of signature-based intrusion detection, the situation did not improve much using these kind of generic signatures. The list of attacks belonging to a certain class of attacks (e.g., TCPDOSAttack) is not complete — this is just a small subset of the methods which have already been identified, but there may be additional ones which have not been discovered yet. This means that the generic signatures approach does not help with detecting previously unknown attacks or variations of known ones, as the exact sub-signatures have to be available for the basic attack instances that should be detected. So although this approach might help in making signatures more maintainable, no previously unknown attack can be identified with this method.

In short one can state that the existing efforts for abstracting from individual signatures are not good enough, as they do not allow to detect new attacks or even variations of existing ones.

4.2 Abstract Signatures

In contrast to the just described methods for increasing the expressivity, *abstract signatures* work in a completely different way. Instead of giving a list of known signatures to a pattern matching engine (which all of the discussed engines do), a model of an attack is used for describing an abstract signature. This model then tries to detect these similarities.

Definition:

A model of an attack is an *abstract signature* if it meets all of the following properties.

- **It can detect a whole class of attacks,** which means that the model should be able to detect not only a few given instances of the attack-class. Attacks belonging to the attack-class addressed by the model and that have been newly created or that are modified versions of known attacks instances should be detected.
- **It has to be update free.** As no attack-class instances are used by the model, an update of the model should not be necessary. Therefore the usual 'update and run' life-cycle of signature-based IDS does not have to performed.
- **It has a very low false positive rate,** as only a very low false positive rate can make an model useful. If a model has a high false positive rate, then the use of the model is voided. System administrators tend to ignore alerts when too many false alerts are reported.

Besides these theoretical properties of abstract signatures there is one extra property of *efficiency* that has to be fulfilled to make an abstract signature usable in practice. Even if an abstract signature has been found, the effort to compute this model is important. If this effort is too high, it does not meet the real-time requirement of intrusion detection systems anymore, voiding its use. As a consequence an algorithm has to be found that is able to compute the required result efficiently.

Abstract signatures therefore combine the advantages of both misuse and anomaly-based systems, ruling out their negative effects. Abstract signatures are able to detect new attack instances of their attack-class and have a very low false positive rate. They cannot be used for detecting completely new attacks, but at least they can be used for detecting modifications of existing attacks, where the attack class is not changed by the modifications. In contrast to anomaly-based systems it is not necessary to build up a profile and continuously train the ID system, because as the similarities of attacks have been identified and coded into the system. The profile in anomaly-based systems represents legitimate behavior, and only when behavior with strong deviations from this profile is observed, an alert is generated. Like all misuse-based systems abstract signatures specify for which events to search for to detect intrusions.

If an abstract signature has to be generated for a certain attack class, the following procedure should be used for ensuring that the resulting sensor does implement an abstract signature.

1. **Analyze attacks of attack-class.** In the beginning an in-depth analysis of attack instances of a certain attack-class is necessary. The goal of this analysis is the identification of similarities that all attacks instances have.

2. **Build a hypothesis.** After the analysis and the identification of attack instance similarities, a hypothesis about has to be built up.

3. **Verify the hypothesis.** The hypothesis has then to be verified in some way. This can either be done in an experimental way, or by providing a formal proof (i.e., by analyzing protocols, headers etc.). The intent of this step is to show why only attacks have the chosen similarity, and why this similarity does not happen during normal operation. If the hypothesis does not hold, one has to revert to the first step and to identify other similarities.

4. **Design an efficient algorithm.** As soon as the hypothesis has been verified, an efficient algorithm for computing the targeted property from a monitored event has to be found. As runtime is the most crucial attribute of ID sensors, appropriate data structures have to be identified to make the algorithm as fast as possible.

5. **Implement and test sensor.** Finally the implementation of the sensor can be done. For this it is necessary to choose the right domain (either host- or network-based) and the right location (e.g., deploy it at the operating system level or integrate it directly into an application).

Using this methodology for creating abstract signatures it is possible to determine at the earliest point in time that a certain sensor will not live up to the expectations, and therefore allows a resource- and cost-reduced sensor development. Abstract signature development is not an easy task, as there is no guarantee that an abstract signature exists for a certain attack-class.

In the following two sections descriptions of constructed abstract signatures can be found. Each section is subdivided into an analysis of a certain attack-class, a proposal for an abstract signature, a validation of the abstract signature, a description of the implementation and an evaluation.

4.3 Buffer Overflow Detection

Although well-configured firewalls provide good protection against many attacks, some services (such as HTTP, FTP or DNS) have to be publicly available. In such cases a firewall has to allow incoming traffic from the Internet without restrictions. The programs implementing these services are often complex and old pieces of software. This inevitably leads to the existence of programming bugs. Skilled intruders exploit such vulnerabilities by sending packets with carefully crafted content that overflow a static buffer in the victim process. This allows the intruder to alter the execution flow of the service daemon and to execute arbitrary code that he can inject, eventually leading to a system compromise and elevating

the privileges of the attacker to the ones of the service process. Such an attack is called a *buffer overflow exploit.* Recent studies [68] have indicated that these attacks contribute to a large number of system compromises as many daemons run with root privileges.

For existing methods of detecting buffer overflow exploits see [22, 20], [25] or [14]. These systems are capable of detecting buffer overflows attacks against service daemons but only *after* they have performed the exploit and and manifest themselves in abnormal behavior. This has the problem that damage might have already occurred and most of them require a recompilation of the service that should be protected.

4.3.1 Buffer Overflow Exploits

Services are usually operated in a client/server setup which means that clients send request data to the server. The server then computes a corresponding answer from the given input, and eventually returns a reply containing the desired results or an error message. This allows virtually anyone (including people with malicious intents) to send data to a remote server daemon which has to analyze and process the presented data.

The server daemon process usually allocates memory buffers where request data received from the network is copied into. During the handling of the received data, the input is parsed, transformed and often copied several times. Problems arise when data is copied into fixed sized buffers declared in subroutines that are statically allocated on the process' stack. It is possible that the request that has been sent to the service is longer than the allocated buffer. When the length of the input is not checked, data is copied into the buffer by means of an *unsafe C* string function. As the statically allocated buffer is smaller than the input data, parts of the stack that are adjacent to the static buffer may be overwritten - a *stack overflow* occurs.

Unsafe C library string functions (see Table 4.1 for examples) are routines that are used to copy data between memory areas (buffers). Unfortunately, it is not guaranteed that the amount of data specified as the source of the copy instruction will fit into the destination buffer. While some functions (such as strncpy) at least force the programmer to specify the number of bytes that should be moved to the destination, others (such as strcpy) copy data until they encounter a terminating character in the source buffer itself. Nevertheless, neither functions check the size of the destination area.

strcpy	wstrcpy	strncpy	wstrncpy
strcat	wcscat	strncat	wstrncat
gets	getws	fgets	fgetws
sprintf	swprintf	scanf	wscanf
memcpy	memmove		

Table 4.1: Vulnerable C Library Functions

Especially functions that determine the end of the source buffer by relying on data inside that buffer carry a risk of overflowing the destination memory area. This risk is especially high when the source buffer contains unchecked data directly received from clients as it allows attackers to force a stack overflow by providing excessive input data.

The fact that C compilers (such as gcc [24]) allocate both, memory for local variables (including static arrays) as well as information which is essential for the program's flow (the return address of a subroutine call and the corresponding frame pointer) on the stack, makes static buffer overflows dangerous. Figure 4.3 below shows the stack layout of a function compiled by gcc. When an attacker can overflow a local buffer stored on the stack and thereby modify the return address of a subroutine call, this might lead to the execution of arbitrary code on behalf of the intruder.

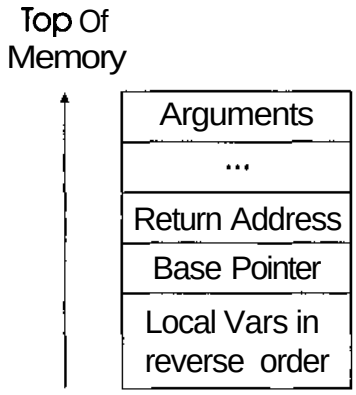


Figure 4.3: Stack Layout

An adversary that knows that a subroutine in the daemon process utilizes a vulnerable function (e.g., strcpy) can launch an attack by sending a request with a length that exceeds the size of the statically allocated buffer used as the destination by this copy instruction. When the server processes his input, a part of the stack including the subroutine's return address is overwritten (see Figure 4.4 below). When the attacker simply sends garbage, a segmentation violation is very likely to occur as the program continues at a random memory address after returning from the subroutine.

A skillful attacker however could carefully craft his request such that the return address points back to the request's payload itself which has been copied onto the stack into the destination buffer. In this case the program counter is set to the stack address somewhere in the buffer that has been overflown when the subroutine returns. The processor then resumes execution of the bytes contained in the request with the privileges of the server process (often with root rights).

The main problem with this technique is the fact that the attacker does not know the exact stack address where his payload will be copied to. Although the intruder can compile and analyze the service program on his machine to get a rough idea of the correct address at the victim's machine, the exact value depends on the environment variables that are set on the victim's machine. When a wrong address is selected, the processor will

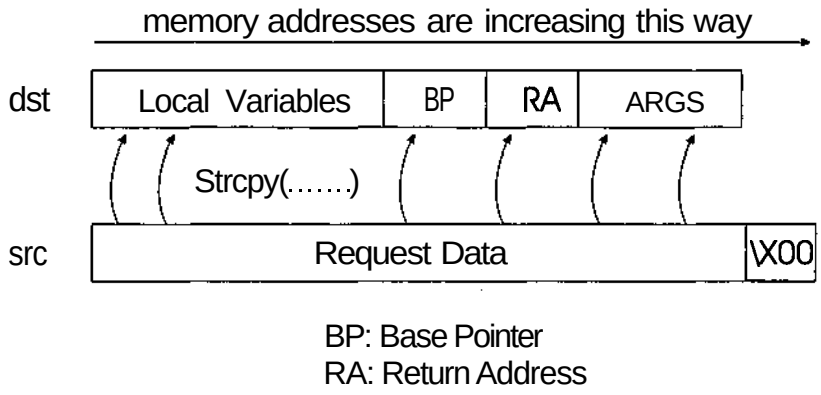


Figure 4.4: Operation of `strcpy(char * dst,char *` src)

start to execute instructions at that position. This is likely to result in an illegal opcode exception because the random value at this memory position does not represent a valid processor instruction (thereby killing the process). Even when the processor can decode the instruction, its parameter may reference memory addresses which are outside of any previously allocated areas. This causes a segmentation violation and a termination of the process.

To circumvent this problem, the attacker can put some special code in front of the exploit code itself to increase the chances of having the faked return address point into the correct stack region. This extra code is called the *sledge* of the exploit and is usually formed by many (a few hundreds are common) NOP (no operation) instructions. The idea is that the return address simply has to point somewhere into this long sledge that does nothing except having the processor move the program counter towards the actual exploit code. Now it is not necessary anymore to hit the exact beginning of the exploit code but merely a position somewhere in the sledge segment. After the exploit code, the guessed return address (RA) (which points into the sledge) is replicated several times to make sure that the subroutine's real return address on the stack is overwritten. A typical layout of a buffer overflow code that includes a sledge is shown in Figure 4.5.

Figure 4.5: Typical Structure of a Buffer Overflow Exploit

Some network-based misuse IDS (such as Snort [66]) try to identify buffer overflow exploits by monitoring the network traffic and scanning packet payload for the occurrence of suspicious bytecode sequences. These sequences are drawn from actual exploits and represent strings such as /bin/sh or operating system calls.

This suffers from the problem that there are virtually infinite variations of buffer overflow exploits that attack different vulnerabilities of the same service or express the same functionality differently. In addition, code transformation techniques such as reordering or

insertion of filling instructions change the signature of the exploit and render the misuse-based detection useless. Some intruders have even started to encode the actual exploit with a simple routine (e.g., ROT-13) while placing the corresponding decode routine in front of the encrypted exploit. When the buffer overflow is executed the decode routine first decrypts the exploit segment and then executes it.

The wide variety of different exploit signatures shifted the focus of these systems to the sledge. Every attack includes a long chain of architecture specific NOP (no operation) instructions that precedes the actual exploit — NOP has a byte representation of 0x90 for the Intel IA32 [30], for other architectures refer to [53].

4.3.2 The Sledge — a similarity of buffer overflow exploits

Unfortunately, the sledge of a buffer overflow exploit can also use opcodes different from NOP causing the signature detectors to fail when these instructions are replaced by functionally equivalent ones. According to [36] there are more than 50 opcodes for the Intel IA32 architecture which are suitable for replacing NOP operations in a one-to-one manner (Table 4.2 below enumerates a few examples).

Mnemonic	Explanation	Opcode
AAA	ASCII Adjust After Addition	0x37
AAS	ASCII Adjust After Subtraction	Ox3f
CWDE	Convert Word To Double-word	0x98
CLC	Clear Carry Flag	Oxf8
CLD	Clear Direction Flag	Oxfc
CLI	Clear Interrupt Flag	Oxfa
CMC	Complement Carry Flag	Oxf5
...	...	...

Table 4.2: Single-Byte NOP Substitutes for IA32

Using these operations (or any combination of those) causes the sledge to behave exactly the same as before, nevertheless its shape can be modified to evade misuse-based ID systems. Notice that it is not possible to encrypt the sledge because it has to be executed before the exploit code (and any decryption routines).

The operations presented in Table 4.2 behave exactly such as the NOP instructions on an Intel architecture in the sense that they are only a single byte long. By considering the fact that modern compilers align variables and data structures on the stack at word boundaries[1], more sledge modifications can be performed.

[1] Word alignment means that the address of any variable allocated on the stack modulo four equals 0

Instead of using single byte instructions without parameters, an intruder can even use multi-byte opcodes with arguments. One just has to make sure that executable code is present at all positions starting at word boundaries. This is necessary because the return address could point to the beginning of any word (i.e., 4 bytes) inside the sledge. This allows the attacker to choose assembler instructions such as the ones depicted in Table 4.3 below or any similar to them. Operations that require an immediate parameter are best suited for this because there is no risk of accessing illegal memory areas (thereby creating a segmentation violation). The attacker just has to use a return address that also points to a word aligned boundary.

Instruction	Bytecode
adc $0x70,%cl	0x80 0xd1 0x70
xor $0x70,$al	0x34 0x70
and 0x70345678, %eax	0x23 0x05 0x78 0x56 0x34 0x70
bsf 0x91 (%eax), %ax	0x66 0x0f 0xbc 0x40 0x91
adc $0x91, 0x70345678(%ebx)	0x83 0x93 0x78 0x56 0x34 0x70 0x91
jmprel 0x37	0xeb 0x37
...	...

Table 4.3: Multi-Byte NOP Substitutes for IA32

The only restriction that remains for the intruder when creating the exploit code and the sledge is that no NULL (0x00) characters may be present. This is due to the fact that a NULL character is interpreted as the end character by many vulnerable C functions. Because the complete attack code has to be copied, the routines may not terminate prematurely. Other than that, the intruder has virtually no limitations in designing his exploit.

Any network-based misuse-system can be easily evaded when such a freedom is given in choosing the layout of the attack. Even anomaly-based systems [41] that base their analysis on the payload of the packet could be fooled. Such systems operate with profiles of a 'normal' request that can be imitated with the means shown above. Network-based anomaly-detectors that consider only protocol information or the flow of traffic fail as well because only a few legal packets need to be transmitted.

While host-based anomaly sensors can notice the effect of a completed buffer overflow and raise an appropriate alarm, this approach is undesirable because of two reasons. First, only successful attacks which cause a corresponding distortion in process behavior are reported. No indication on the number of attempts is given. Second, the system can only react to the attack after it has manifested itself as weird behavior. Potential damage that has been inflicted before the ID reacts cannot be prevented (e.g., deletion/modification of files).

As could be seen above, the attacker has various possibilities of shaping and cloaking his exploit. Nevertheless, a sledge has to be present at the beginning of the exploit to make

the exploit work deterministically. We also saw that the sledge cannot be encrypted, as it is the first element of the exploit that is executed. The shape of the sledge can be varied vastly, as shown in Table 4.3, and as it can contain relative and even absolute jumps. So no exact statement about the content of the sledge can be made, as it can be customized vastly.

The only property that has to remain valid through all these transformations and substitutions is that the sledge has to be *executable.* When jumped to a random position within the sledge, the program flow will lead to the exploit. As no possibility exists to masquerade this behavior without voiding its use, this attribute of buffer overflow exploits can be used for their reliable detection. As the payload cannot be safely executed, the execution is only simulated, which we call *abstract execution.*

4.3.3 Abstract Execution

The idea of our approach, which has been published in [80], is to focus on the executability of the sledge (which cannot be prevented without breaking the attack) by means of abstract execution. Before we are able to formulate and verify a hypothesis, we have to define some notions and metrics. At first, the notions of valid instructions and valid instruction chains are introduced.

The following two properties that classify a sequence of bytes executed on behalf of a certain process are used to define *abstract execution.*

- **Correctness:** A sequence of bytes is correct, if it represents a single valid processor instruction. This implies that the processor is able to decode it. The byte sequence consists of a valid opcode and the exact number of arguments needed for this instruction (and none more). Otherwise, the sequence is incorrect.

- **Validity:** A sequence of bytes is valid if it is correct and all memory operands of the instruction reference memory addresses that the process which executes the operation is allowed to access. A memory operand of an instruction is an operand that directly references memory (i.e., specifies an address in the memory area of the process). Validity is important because references to non-accessible memory addresses will be detected by the operating system resulting in the immediate termination of the process with a segmentation violation. A correct instruction without any (memory) operands (e.g., `NOP`) is automatically valid. We also call a valid byte sequence a valid instruction.

Definition:

> A sequence of bytes is *abstract executable* if it can be represented as a sequence of consecutive valid instructions.

Usually two different types of instructions exist. First there are instructions that are executed and afterward the program flow is continued at the instruction following the current one. Instructions of this type are called *non-jumping instructions.* The second class of instructions has the duty of explicitly changing the program flow, which is not necessarily continued at the instruction following the current one. These instructions belong to the class of*jumping instructions,* (e.g., call, jmp, jne). As the semantics of both types are completely different but are very important for measuring the executability, they have to be treated in a different way. Therefore we partition the pool of processor instructions into two sets. One contains all instructions that alter the execution flow of a process (i.e., operations that modify the program counter, jumping instructions) while the other set consists of the rest (non-jumping instructions). The elements of the first set are called *jump* instructions. A sequence of valid instructions can be decomposed into subsequences that do not contain jump instructions. Such subsequences are called *valid instruction chains (VIC).* An instruction chain ends with a jump instruction, at the end of the buffer or at bytes that are not abstract executable. The length of an instruction chain is equal to the number of instructions that it consists of.

An important metrics is the **execution length** of a sequence of valid instructions. The basic idea is that the execution length combines the lengths of instruction chains that are connected by jump instructions. It can be computed for a byte sequence using the following algorithm.

Algorithm: *Abstract Execution Length* of a Byte Sequence

> The algorithm expects two input arguments, a byte sequence seq and a position pos in this sequence. It uses an auxiliary array visited to mark already visited blocks whose elements are initialized with `false`. The return value is a positive integer denoting the execution length starting at position pos.

1. Initialize the value of L to 0.
2. When the instruction at pos is invalid, return 0.
3. When the instruction at pos has already been visited, a loop is detected and 0 returned.
4. Find the instruction chain starting at pos and calculate its length *L.* In addition, mark the corresponding entry of the first byte of the instruction in the visited buffer.
5. When the instruction chain ends with invalid bytes, return *L.*
6. Otherwise, the chain ends in a jump instruction. When the target of the jump is outside the byte sequence seq or cannot be determined statically, return $L + 1$.
7. When the jump targets an operation at position target that is inside the sequence, call the algorithm recursively with the position set to target and assign the result to L'.

8. When the jump is unconditional, return $L + L'$.

9. Otherwise, it is a conditional jump. Call the algorithm recursively for the continuation of the jump — i.e., set the position to the operation immediately following the jump instruction and assign the result to L''. Then determine the maximum of L' and L'' and assign it to L_{max}. Then return $L + L_{max}$.

The execution length describes the number of instructions that could be executed for sure by the CPU when started at a certain position and also takes into account jumping instructions.

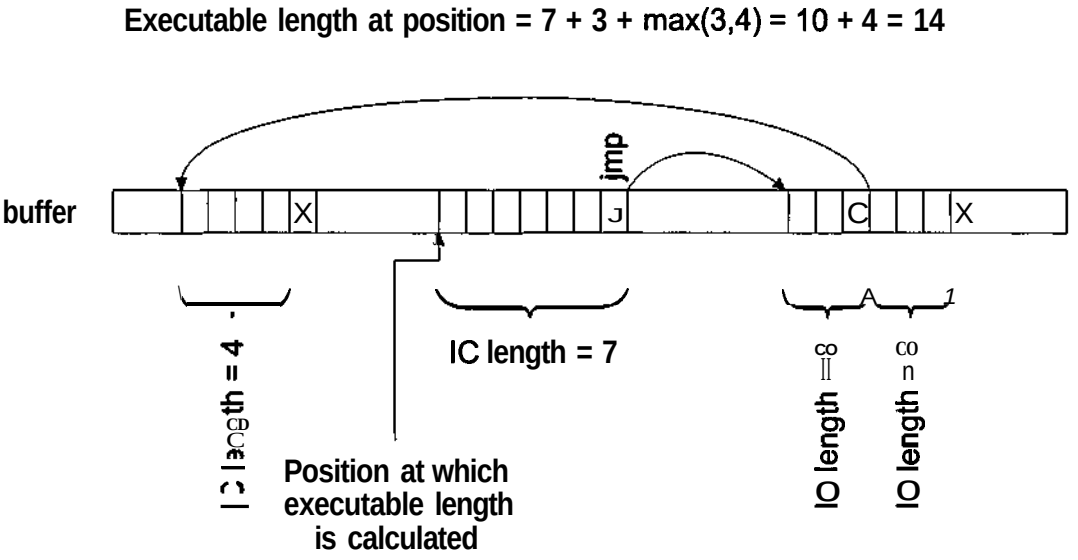


Figure 4.6: Determining the Execution Length at a Certain Position

Figure 4.6 shows an example of the algorithm while determining the execution length at a certain position.

For identifying sledges within payloads, the maximum of all abstract execution lengths is interesting.

Definition:

> The *maximum execution length (MEL)* of a byte sequence is the maximum of all abstract execution lengths that are calculated by starting from every possible byte position in the sequence. It is possible that a byte sequence contains several disjoint abstract execution flows and the MEL denotes the length of the longest.

4.3.4 The Hypothesis and its Experimental Verification

The hypothesis that has been used for developing the buffer overflow detection mechanism reads:

> Normal requests are not as *abstract executable* as requests that contain buffer overflow exploits, i.e., the MEL of normal requests is much lower than the MEL of attack the requests containing

$$MEL(normalrequests) << MEL(attackrequests) \quad (4.1)$$

A formal proof for the hypothesis cannot be given, as the payloads of requests can contain merely arbitrary text based data. Sledges of buffer overflow exploits have to consist of byte sequences that form valid instruction chains. As these instructions are not uniformly distributed over all possible byte combinations but instead form only islands within this domain, an intuitive reason for this is given. In order to test whether the hypothesis holds, we measured the MELs of a large set of normal requests to various services as well as the MELs of attack requests and compared them to each other.

Execution Length of HTTP Requests

In order to estimate the maximum execution length of regular HTTP requests, we calculated the MEL for service requests targeted at our institute's web server. Only successful requests that completed without errors have been included in our test data set. We also manually removed attack requests to avoid that buffer overflow exploits distort the data set. An additional ID system has been deployed to verify that assumption. 117228 server requests which we have been captured during a period of 7 days have been processed. The resulting MELs are shown in Figure 4.7 below.

Only 350 requests had a MEL value of 0 meaning that they did not contain a valid instruction at all. Most of the packets showed a maximum execution length of 3 and 4 (33211 and 31791 respectively) with the numbers decreasing for increasing lengths. The highest maximum instruction length that has been encountered was 16 which appeared for a total of 14 HTTP queries. As expected the numbers indicate that the MELs for regular requests are short.

Additionally we analyzed the data produced during the first week of by the MIT Lincoln Labs for their 1998 DARPA Intrusion Detection Evaluation [46]. We extracted all HTTP requests that have been sent and calculated the MELs. The results are listed in Table 4.4.

As can be seen easily, there is an upper limit of the MELs of normal HTTP traffic, as the longest MEL is 26 and only occurs one time. Most of the requests have a MEL smaller than 10. This clearly supports the hypothesis.

MEL	Monday	Tuesday	Wednesday	Thursday	Friday
0	169983	3533928	194009	190702	111712
1	0	1	0	0	0
2	0	1	0	0	0
3	0	1	0	0	0
4	0	0	0	0	0
5	1039	1742	1287	972	273
6	1575	2346	2402	1905	618
7	6406	7188	6998	6300	2327
8	6701	7716	6653	5948	2600
9	6929	8828	7660	6975	2326
10	5035	7086	5937	5990	2095
11	3108	4278	3435	3255	1290
12	2165	3321	2897	2169	882
13	1438	1694	1565	1799	531
14	492	1056	909	916	467
15	420	448	911	754	141
16	194	224	140	208	61
17	92	143	187	93	85
18	66	44	80	42	33
19	26	47	42	38	51
20	6	47	9	7	9
21	5	17	36	18	1
22	2	5	1	3	0
23	6	90	2	8	0
24	5	5	1	1	1
25	16	1	11	0	0
26	0	1	0	0	0
> 26	0	0	0	0	0

Table 4.4: Maximum Execution Length Distribution in DARPA'98 HTTP Traffic, Week 1

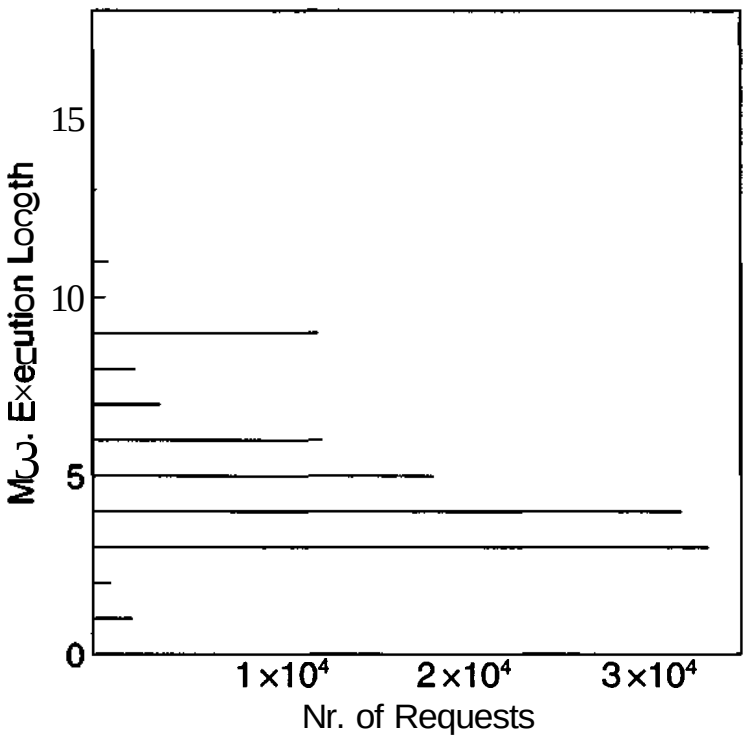


Figure 4.7: Maximum Execution Length of Regular HTTP Requests

Execution Length of DNS Requests

We performed a similar experiment as explained above on DNS data. We captured all the DNS traffic (from the inside and from the outside) to our DNS server during a period of one week. We collected 75464 requests and calculated the MEL on each of these.

As shown in Figure 4.8 the maximum execution length distribution has its peak at 4 with 58557 request. In descending order the MELs of 5 and 3 follow with 6531 and 5500 requests, respectively. The maximum MEL found in our sample data is 12 which has been present in only 4 requests. Therefore the maximum of all MELs of the measured DNS requests is even lower then the maximum of measured HTTP requests.

We performed the same measurements on the DARPA'99 traffic on the DNS request data as before on the HTTP traffic. Table 4.5 lists the maximum of all MELs that have been observed for the respecting day. The distribution is nearly the same as for HTTP, i.e., most of the requests have a MEL smaller than 10 and the upper bound is very low, even lower than for HTTP requests.

	Monday	Tuesday	Wednesday	Thursday	Friday
Maximum MEL	10	9	9	9	9

Table 4.5: Maximum MELs in DARPA'98 DNS Traffic, Week 1

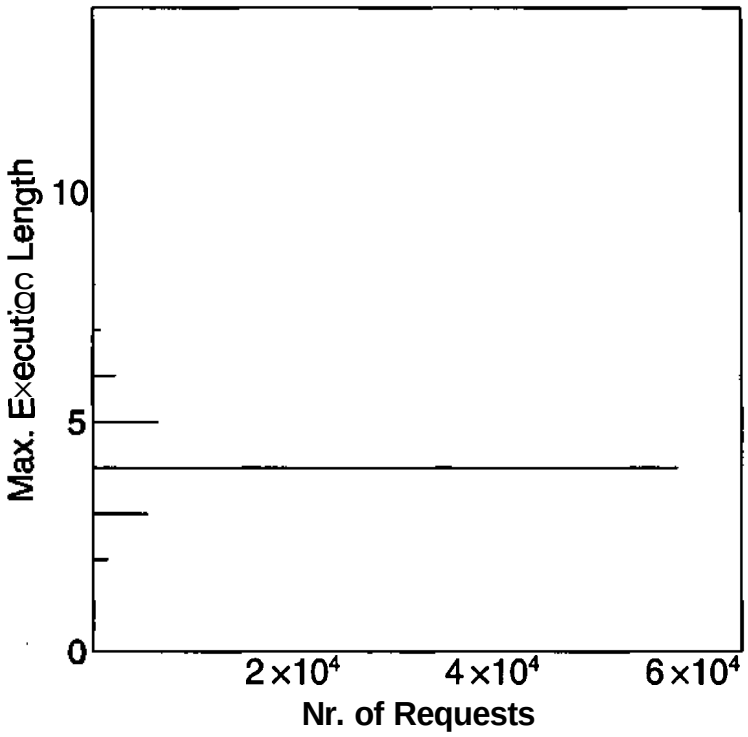


Figure 4.8: Maximum Execution Length of Regular DNS Requests

Execution Length of FTP Requests

Finally we measured the MELs of the FTP requests that can be observed in the DARPA'99 first week traffic. We only measured traffic directed to port 21, as this is the channel that is used for exchanging control and status data between an FTP server and client. This is also the channel that is used by an attacker for sending the exploit code to the server. We performed the same MEL measurements on FTP request data that has been sent from clients to servers. Table 4.6 lists the maximum of all MELs that have been observed for the respecting day. Similar to HTTP and DNS the majority of requests have a very small MEL.

	Monday	Tuesday	Wednesday	Thursday	Friday
Maximum MEL	13	16	12	12	12

Table 4.6: Maximum MELs in DARPA'98 FTP Request Traffic, Week 1

Execution Length of Exploits

Now that we have seen that all requests to HTTP, DNS and FTP services had a very tight upper MEL, we have to investigate the MELs of exploits. As already explained buffer overflow

exploits usually have a sledge that consists of several hundred bytes, which shows that the relation 4.1 truly holds. In order to support our claim that buffer overflow attacks contain long valid instruction chains, a number of available exploits have been analyzed. We have chosen buffer overflow exploits against the Internet Information Service (IIS) (the web server from Microsoft), BIND (a UNIX DNS server) and WU-FTP (a UNIX FTP server), all from [68]. Although our prototype has been developed and tested with a web server, attack code against a different service daemon has been evaluated to show the applicability of our approach to other areas as well. The results of this evaluation are listed in Table 4.7.

Exploit	Max. Execution Length (MEL)
IIS 4 hack 307	591
JIM IIS Server Side Include overflow	807
wu-ftpd/2.6-id1387	238
ISC BIND 8.1, Bugtraq ID 1887	216

Table 4.7: Maximum Execution Lengths of Exploits

According to the table above, the maximum execution lengths of requests that contain buffer overflow exploits is significantly higher than those of normal queries. This observation supports our hypothesis (4.1) that MELs of attacks are much higher than MELs of legitimate requests. This knowledge can now be used for detecting intrusions, as an abstract signature has been found.

4.3.5 Detecting Buffer Overflows

We present an in-line intrusion detection sensor which implements an abstract signature that can a-priori prevent malicious code from being executed by analyzing the content of service requests at the application level. These services usually operate in a client/server setup where a client machine sends a request to the server which returns a reply with the results. Our detection approach distinguishes normal request data from malicious content (i.e., buffer overflow code) by performing abstract execution of payload data contained in client requests. In the case of detecting long 'instruction chains' of executable code (see Section 4.3.3) a request can be dropped before the malicious effects of the exploit code are triggered within vulnerable functions (and maybe detected by another ID system afterwards).

Following the definitions above, we expect that requests which contain buffer overflow exploits have a very high MEL. The sledge is a long chain of valid instructions that eventually leads to the execution of the exploit code. Even when the attacker inserts jumps and attempts to disguise the functionality of this segment, the execution length is high. In contrast to that, the MEL of a normal request is comparatively low. This is due to the fact that the data exchanged between client and server is determined by the communication protocol and has a certain semantics. Although parts of that data may represent

executable code, the chances that random byte sequences yield a long executable chain is very small.

The idea is that a *static threshold* can be established that separates malicious from normal requests by considering requests with a large MEL as malicious while those with a small execution length as regular. One question that immediately arises is to which value to set the static threshold. It should be clear that the number has to be in between the measured MELs of normal requests and the MELs of exploits. To keep the false positive rate very low it is necessary to prevent that regular requests are identified as malicious — the threshold should be placed with a certain distance to the maximum expected MEL. On the other hand, it should be nearly impossible for attackers to create an exploit that deterministically works and has a MEL lower than the established threshold, therefore the distance of the threshold should not be near the observed MELs of known exploits. We used the following formula for determining the static threshold.

$$difference = min(MEL_{exploit}) - max(MEL_{normal}) \tag{4.2}$$

$$threshold = max(MEL_{normal}) + min(difference/2, 10) \tag{4.3}$$

This ensures that the static threshold is placed very near the maximum expected MEL of normal requests, but is far away from the MELs of already observed exploits.

The sledge has to be executable in order to fulfill its task, therefore a simple test can be utilized to find long executable chains. The test is placed at a position where the request has been completely received, but where not further processing has been done yet. In this stage, requests containing buffer overflow exploits cannot make use of their hostile effects, but they can be tested for malicious code. Requests are analyzed immediately and checked for long MELs. This enables the system to drop potential dangerous requests which have a MEL larger than the established static threshold and to let normal requests to proceed. Dropping potential dangerous requests prevents the service process to be affected, as the request payload never reaches vulnerable functions, which cause the buffer overflow exploits to unfold their malicious capabilities. We have chosen to place our sensor at the application layer to circumvent the problem of encrypted network traffic faced by network-based IDS — therefore encryption is not an issue for our approach.

The following observation allows an improvement of the test algorithm that has to determine the MEL of requests. According to the definition of the maximum execution length, all positions in the request's byte sequence could potentially serve as a starting point for the longest execution flow. However, if the MELs of normal requests and exploits differ dramatically, it is not necessary to search for the real maximum length, i.e., determine the execution length at every possible starting point. It is sufficient to choose only some random sample positions which are uniformly distributed within the byte sequence and to calculate the execution length from these positions. Instructions that have been visited by earlier runs of the algorithm are obviously ignored. The rationale behind this improvement is the fact that it is very likely that at least one sample position is somewhere in the middle of

the sledge leading to a tremendously higher MEL than encountered when checking normal requests.

Anyway, not every byte serves as a starting point for the execution length calculation. As already mentioned in this chapter buffers are aligned at word boundaries and the return address can only point to word boundaries too. Therefore only the bytes being at the beginning of words serve as starting points for the MEL calculation, effectively dividing the number of starting points by four.

4.3.6 Implementation

We implemented the algorithms for determining a single execution length and for choosing reasonable sample points in the byte sequence of a request in C. Because the recursive procedures are potentially costly, the main focus has been on an efficient realization. As every request needs to be evaluated, the additional pressure on the server must be minimized.

An important point is the decoding of byte sequences to determine the correctness and validity of instructions. As data structure we have chosen a static Trie for storing all supported processor instructions together with the required operands and their types.

A definition of a Trie has already been given in Chapter 3. The words which are stored in the Trie are opcodes in our case. Each byte of an opcode is stored in the Trie at a different level. The first byte of an opcode is stored at the root node (first level) of the Trie together with a pointer to a second-level Trie node that stores the continuation of all the opcodes starting with this first byte. We store all supported opcodes of the processor's instruction set in the Trie to enable rapid decoding of byte sequences. The leaf nodes hold information about the number of operands for each instruction together with their types (immediate value, memory location or register). This enables us to calculate the total length of the instruction at runtime by determining the necessary bytes for all operands.

It is important to notice that different instructions can be of different length, therefore a hash table is not ideally suitable. Currently, only the Pentium instruction set [30] has been stored in this Trie, but no `MMX` and `SIMD` instructions are supported.

Figure 4.9 shows a simplified view of our Trie. The opcodes for the instructions AAA (opcode 0x37), ADC (opcode 0x661140 — add with carry the ax register to the value of the register indirect address determined by eax and the one byte operand), ADC (opcode 0x80d1 — add with carry a one byte value to the `cl` register) and CMP (opcode 0x80fc — compare the immediate value with the register ah) have been inserted.

The algorithms used to determine an approximation of the MEL of HTTP requests have been integrated as a module into an Apache `1.3.23` web server. During the startup of the server, the Trie is filled and a function to check the request is registered as a post_read_request procedure. The Apache configuration file has been adapted to make sure that our module is the first to be invoked.

Each time a request arrives at the HTTP server, our subroutine calls the URL decoding routine provided by Apache and then searches for executable instructions in the resulting byte sequence. It is necessary to decode the request first to make sure that all escaped characters are transformed into their corresponding byte values. The module uses a defin-

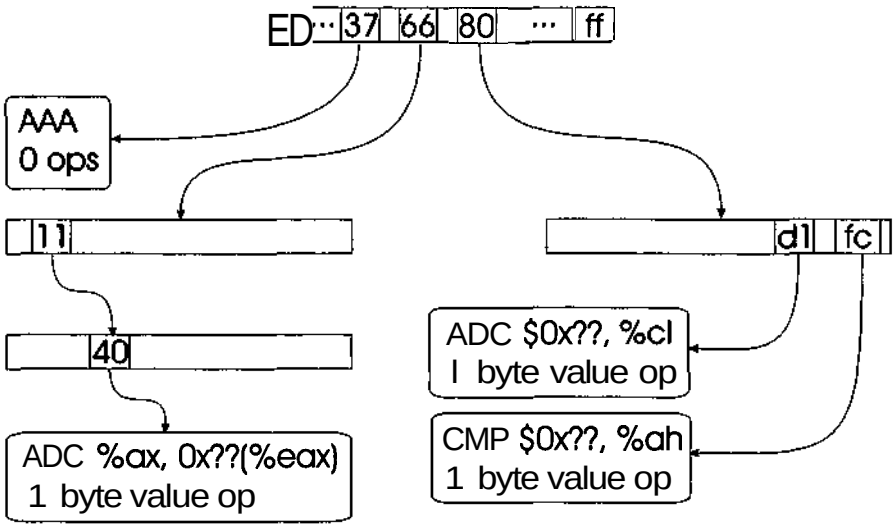


Figure 4.9: Storing Instruction Opcodes in a Trie

able threshold and stops the test immediately when a detected execution length exceeds this limit. We do not calculate the MEL of the request because of performance reasons. Instead we chose to calculate the execution length at equally distributed positions within the request, only starting at the first bytes of words.

For our first prototype, we simply select a 'reasonable' magic number between the maximum value gathered from the set of normal requests (26) and the minimum among the evaluated exploits (216). Because an attacker might attempt to limit the MEL by choosing a shorter sledge, the value should stay closer to the maximum of the normal requests. We decided to select 35 for the deployment of the probe. This leaves enough room for regular requests to keep the false positive rate low and forces an intruders to reduce the executable parts of his exploit to a length less than this limit to remain undetected. Such a short sledge nevertheless seriously impacts the attacker's chances to guess an address that is 'close enough' to the correct one to succeed.

4.3.7 Performance Results

To evaluate the performance impact of our module on the web server, we used the WebSTONE [86] benchmark provided by Mindcraft. WebSTONE can simulate an arbitrary number of clients that request pages of different sizes from the web server to simulate realistic load. It determines a number of interesting properties that are listed below.

- average and maximum connection time of requests
- average and maximum response time of requests
- data throughput rate

The *connection time* is the time interval between the point when the client opens a TCP connection and the point when the server accepts it. The *response time* measures the time between the point when the client has established the connection and requests data and the point when the first result is received. The *data throughput rate* is a value for the amount of data that the web server is able to deliver to all clients.

The connection and response time values are relevant for the time a user has to wait after sending a request until results are delivered back. These times also characterize the number of requests a web server is able to handle under a specific load. The data throughput rate defines how fast data can be sent from the web server to the client. Because clients obviously cannot receive replies faster than the server is sending them, this number is an indication for how long a client has to wait until a request completes.

Our experimental setup consists of one machine simulating the clients that perform HTTP requests (Athlon, 1 GHz, 256 MB RAM, Linux 2.4) and one host with the Apache server (Pentium III, 550 MHz, 512 MB RAM, Linux 2.4). Both machines are connected using a 100 Mb Fast Ethernet. WebSTONE has been configured to launch 10 to 100 clients in steps of 10, each running for 2 minutes. We did only a measurement of static pages, so no tests involved dynamic creation of results.

We measured the connection rate, the average client response time and the average client throughput for each test run with and without our installed module. The results are shown in Figures 4.10, 4.11 and 4.12. The dotted line represents the statistics gathered when running the unmodified Apache while the solid line represents the one with our activated module.

As can be seen above, the connection rate has dropped slightly when our sensor is activated. The biggest difference emerged when 50 clients are active and a value of 494,2 connections per second versus 500,7 connections per second with the unmodified Apache has been observed. While this maximum difference is 6.5 connections per second (yielding a decrease in the client connection rate of about 1.4 %), the average value is only 2,4 (about 0.5 %).

There has been no significant decrease in the average response time. Both lines are nearly congruent with regards to the precision of measurements.

The client throughput decreased most with 10 active clients when it dropped from 75,90 Mb per second to 73,70 Mb per second. This is an absolute difference of 2,2 Mb per second (about 2,9 %). On average, the client throughput only decreased by 0,8 Mb per second (about 1,05%).

The Trie consumed about 16 MB of memory during the tests. While this seems to be a large number at first glance, one has to take the usual main memory equipment of web servers into account where a Gigabyte of RAM is not uncommon. In addition, this data structure makes very fast tests possible and is a classical trade-off in favor of speed.

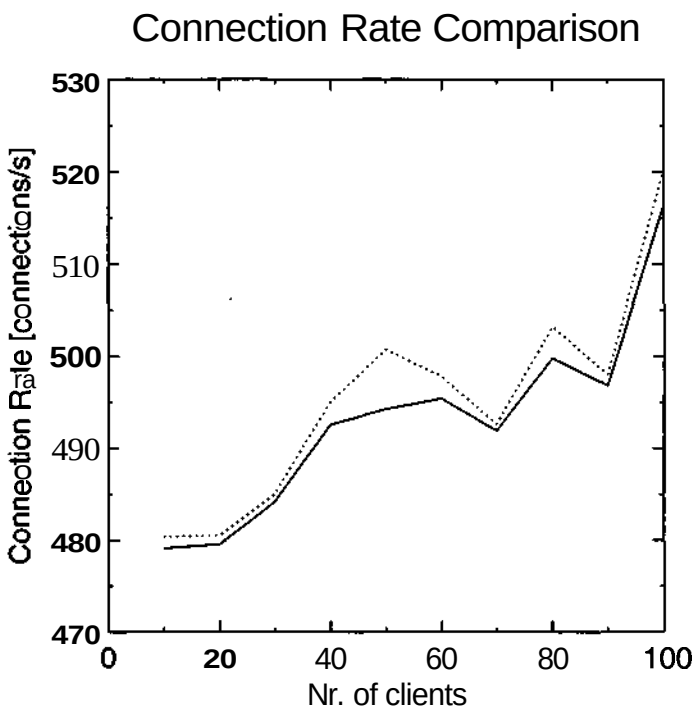


Figure 4.10: Client Connection Rate

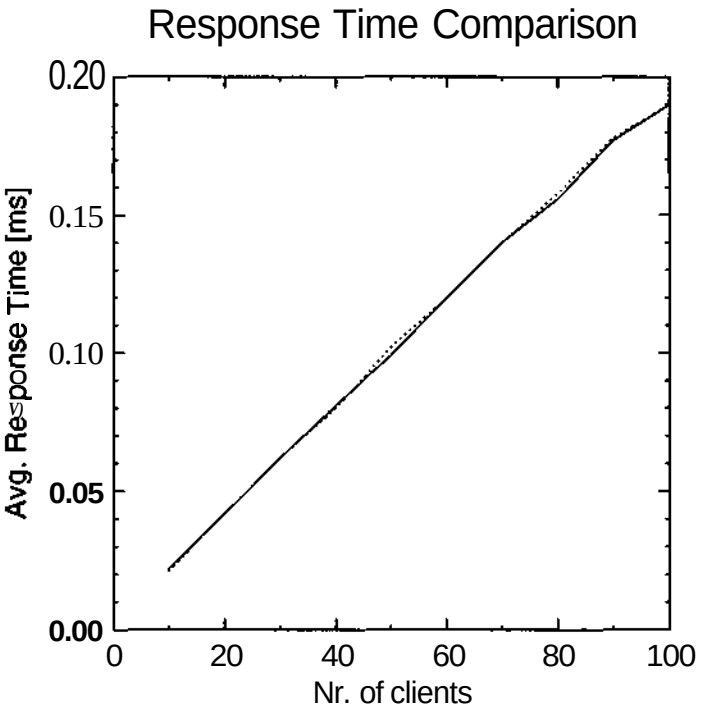


Figure 4.11: Average Response Time

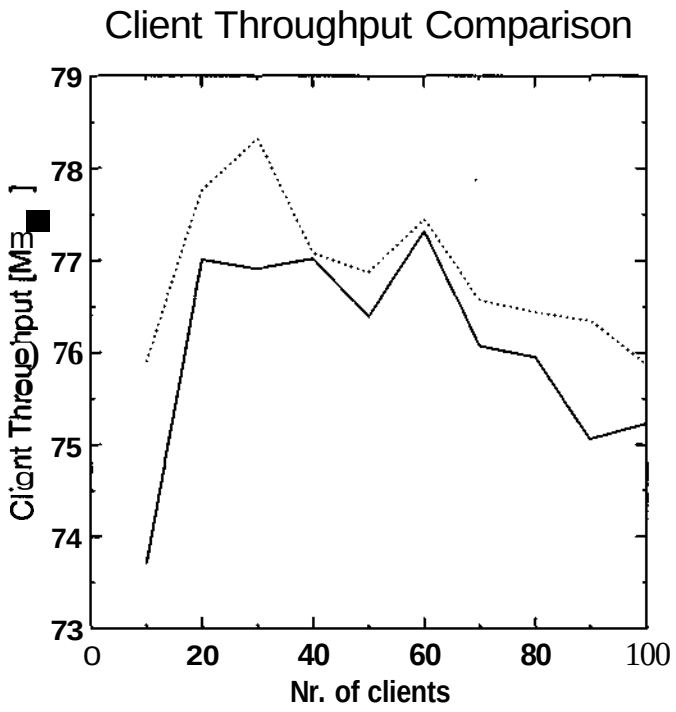


Figure 4.12: Client Throughput

4.4 Worm Detection

In the past few years, many vulnerabilities of wide-spread software services, often running on publicly accessible hosts, have been discovered. These vulnerabilities allow hackers to gain access to those machines, thereby compromising their security. When a vulnerable service is deployed on large numbers of publicly accessible hosts, software tools (called *worms)* can automate the task of intruding a machine and spreading to new locations. This section represents the work that has been published in [81].

Definition:

> A *worm* is a program that can run independently and can propagate a fully working version of itself to other machines. It is derived from the word *tape-worm,* a parasitic organism that lives inside a host and uses its resources to maintain itself [72].

Worms can make use of buffer overflow exploits or can exploit simple programming mistakes and vulnerabilities to propagate. Once they have transferred the data comprising the worm to the victim machine, the victim's machine is forced to execute this code. This causes the now infected machine to behave in the same way as the machine it came from. Worms consume a large amount of bandwidth by attempting to find and infect other vulnerable machines and to compromise the security of these hosts. Because of the fact that worms are most of the time implemented as directly executable code, they are

processed at very high speed. This and the high transmission bandwidths allow a worm to quickly spread over a network. Usually, it is too late when one manually detects the presence of a worm — in most cases the whole network has already been infected. The success of worms is based on the fact that they come up very quickly, faster than somebody can react manually. In general new worms cannot be identified until an analysis has been made of their code — therefore no signatures are available which can be used for them. Even if an analysis has been made, the vulnerable services have to be patched to make them immune against the worm. Once a patched version is available, the individual deployed systems have to be updated to this patched version. As this procedure starting from the analysis of a worm to the completion of all updates takes a very long time compared to the time the worm needs for spreading to a new host, this procedure is not suitable for stopping a new worm. Even if the patch for a vulnerable service would be created instantly, the administrators of the individual sites running this service would have to update their service, which is a manual task.

Another problem is that the behavior (and damage functionality) of a worm often cannot be predicted by simply monitoring its activity over a short period of time. Hidden features might exist and worms often use a wide variety of methods for spreading, having several exploits available to target different services. Recently worms that not only target one service came up, but that can exploit vulnerabilities of several services. Nimda [56] for example is a worm that uses various possibilities for spreading. The worm sends itself out by email, searches for open Windows network shares, attempts to copy itself to un-patched or already vulnerable Microsoft IIS web servers, and is a worm infecting both local files and files on remote network shares. More worms of this kind are expected to be coming up soon.

In the best case, worms only perform a denial of service attack against the infected network by scanning it for vulnerable services and attempting to infect other machines. Unfortunately, worms can carry arbitrary pieces of software with them, even remote management tools that allow third parties to access the compromised host. Such a combination of a worm and a remote management tools such as Netbus of Back Orifice not only wastes valuable resources but threatens confidentiality. Remote management tools allow to execute arbitrary commands on a host allowing others to sniff information such as passwords, credit card numbers or confidential documents from all kind of devices (e.g., keyboard, hard disc, local network traffic, etc.).

As a worm is not only a nuisance to the system administrator but can have a serious impact on system security, a worm infection can be considered as intrusive behavior.

The problem with misuse-based systems is that they require a signature to find attacks — this is usually only the case when a certain exploit is wide-spread and well known. If an attack is performed against a service with an exploit whose signature is unknown, the signature-based IDS does not recognize the malicious behavior and therefore is unable to report an alarm or perform some active countermeasures, although an attack has been performed. Because of the fact that every worm can make use of different vulnerabilities, there exist a vast number of different variations, causing that a unique signature or pattern cannot be created

A well-known intrusion detection system that explicitly deals with the problem of detecting worms is GrIDS [73], see Chapter 2 for details about GrIDS. The aim of this system is to be able to handle a large number of hosts as their intended target are large enterprise networks. Although GrIDS offers many interesting features and is a step into the right direction, it still suffers from several disadvantages. First, their detection is aimed at large scale detection of worms. This process requires data exchange between many nodes in different networks which results in a considerable bandwidth overhead. To make this mechanism scalable, a data reduction scheme has been introduced which causes valuable information to be lost. Second, their worm detection is based on tree shaped patterns with branches at certain nodes (i.e., multiple connections originating at a certain machine). Nevertheless, such branches do not necessarily have to occur in a connection pattern of a worm (although they often do). Third, the worm detector does not include the packet payload into the correlation process. GrIDS only considers events such as the establishment or closing of a connection or takes very specialized information into account such as the user name that was used for a login. Using such a reduced event base makes correlation possible, but does not make full use of the available information. Detecting similarities in the payload of different packets can enable the system to increase the certainty that observed connections really represent a worm. This would allow the system to lower the false positive rate. Fourth, GrIDS does not include any response mechanism in the case that a worm has been detected. This is a major shortcoming, because by utilizing available components such as firewalls, further damage could be prevented.

The intent of this section is to tackle these issues by developing a model that protects the uncompromised machines in a network by quickly. The behavior of a spreading worm is determined and automatic response mechanisms are used to stop the worm from infecting uncompromised ones in near real-time.

4.4.1 Analysis

Connection patterns that indicate spreading worms have to follow the two properties listed below:

- similarity of connection patterns, and
- causality of connection patterns

The *similarity of connection patterns* describes the fact that a worm exhibits similar behavior when it attempts to spread from node to node. This is caused by the fact that a worm only contains a certain number of modules that can launch different attacks against vulnerable services. It is very likely that a worm attempts to repeatedly exploit the same vulnerability at different machines, especially because it is often the case that all hosts of a network are all running the same version of a certain service. Even when different modules are at hand, they are limited in number. Therefore it is justified to assume that there will be detectable similarities in the connection patterns. Nimda was one of the first worms that made use of multiple vulnerabilities.

The *causality of connection patterns* means that a certain connection pattern or event depends on the occurrence of another, preceding event. For example, it is obvious that a node has to be infected first, before it starts to act in an infected manner. Although such causality of events or patterns is not a sure sign of an intrusion, it can provide a strong indication. Notice that through causality property a temporal relationship is established between connections. A connection can only cause another one, if it happened before. An interesting situation is the observation that the destination node of a certain connection opens a similar connection to another host after a short period of time. If the similarity of connection patterns is correlated with the causality of connection patterns, it is possible to reliably identify worms.

With very few exceptions services do not send out the data that they just received on the same network interface card. Exceptions are for example DNS servers that try to resolve requests from a client which they cannot handle. In this case, the query has to be forwarded and causes a very similar connection to another machine. Routing services also behave in the same way of sending out data that they just received — but the major difference to the DNS service is that the outgoing connection is performed on another interface — so the connections are visible in different networks, making it impossible to observe the same payload twice in the same network.

Causal relationships are of interest because they help to dramatically reduce the search space for pattern matching algorithms as only a few connections that happen after a certain event have to be taken into account.

Following these two properties, the characteristics of worm behavior are defined. The occurrence of events are evaluated according to these principles and are assigned a certain severity.

Before a hypothesis can be formulated and verified, the necessary notions have to be defined. We will introduce the concepts of *connection chain* and the *trail* of a node. After that, the hypothesis is set up and verified experimentally.

4.4.2 Definitions

To make the explanation of the pattern evaluation algorithm easier to follow, we first introduce some definitions.

Definition:

> A *connection* is a tuple (timestamp, srcHost, srcPort, dstHost, dstPort and data) representing a successfully opened TCP connection at time timestamp from machine srcHost, srcPort to machine dstHost, dstPort. data does not contain all bytes that have been exchanged, but only the first z ones (where z is a configurable number).

Definition:

> A *connection-set* is a set of connections where each connection is assigned a unique number. The numbers are assigned in the order the connections occur.

Definition:

A *chain* is a subset of a connection-set. For the elements *Ci* ($1 < i < l$) of a chain of length /, $l > 1$ the following properties hold.

- For all connections c_i and c_{i+p} of the connection-set, the property $timestamp(c_i) < timestamp(c_{i+p})$ holds (with $p > 0$).
- For each connection c_i and $i > 1$, the destination of the connection c_{i-1} is the source of *Ci.*

Informally a chain represents a number of TCP session establishments that are sent consecutively from one host to another. Notice that a chain consists at least of two connections. Such a chain is created for example by a user that remotely logs in at one host and from there logs in on another host. A telnet chain is a chain where the remote login service telnet has been used. An outgoing connection from a host is an element of the chain only if another host opened a connection to the mentioned host before. Loops (the same host being present twice in a chain) are allowed as the timestamp allows an exact ordering of connections.

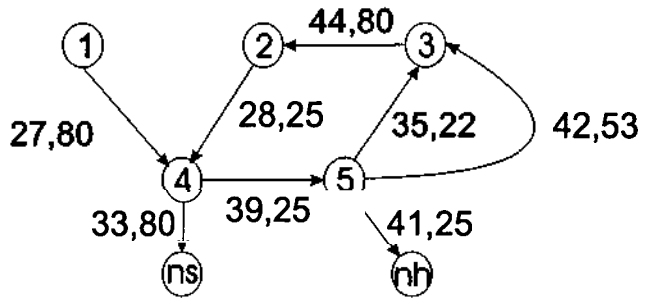
Example Assume that the following connection set can be observed in a network.

connection number	time	src	srcport	dst	dstport	content
1	27	1	1231	4	80	GET\..\..
2	28	2	8329	4	25	...
3	33	4	2215	ns	80	GET\..\..
4	35	5	9782	3	22	GET index.h
5	39	4	11251	5	25	contentA
6	41	5	5214	nh	25	contentA
7	42	5	28315	3	53	GET\\..
8	44	3	18763	2	80	contentA
...	...	...	...	...	...	

ns and *nh* have the meaning that the targeted entity is not existing. *ns* means that no service is listening at the port this connection was opened to while *nh* means that the targeted host does not exist. The shape of the connection graph together with the timestamps and destination ports of every individual connection is shown in Figure 4.13. For this scenario, the chains are listed right to the graph. Each chain consists of the individual connections, which are temporally ordered and separated using '-'. Connections 5 and 4 do not form a chain, as the timestamp condition (which is introduced by causality) is violated.

Definition:

The *trail* of a node N consists of all longest chains that end at node N.



Chains (connection numbers)
1-5
5-7
1-5-7
7-8
5-7-8
1-5-7-8
2-5
2-5-7
2-5-7-8
4-8

Figure 4.13: A Sample Connection Graph and the Associated Chains

Node	Trail
1	{}
2	{4-8, 1-5-7-8, 2-5-7-8 }
3	{ 1-5-7, 2-5-7 }
4	{}
5	{1-5,2-5}

Table 4.8: Trails of the Connection Graph

Table 4.8 shows the trails of the individual nodes of the connection graph.

These definitions allow us to formulate a hypothesis and to develop an abstract signature from it.

4.4.3 The Hypothesis and its Experimental Verification

The hypothesis that has been used for developing the worm detection mechanism reads:

> *Worms show a regularity in their connection patterns — the payload of connections that are opened to other nodes is likely to be very similar or even the same as the payload that could have been observed in the trail of the infected host. In the normal case, such a behavior cannot be observed, but if a host gets infected, he starts to behave in this way.*

This hypothesis is motivated by the fact that worms have to be small to ensure their success and to make them spread quickly. Therefore only the necessary routines find their place in the worm's code, everything unnecessary is let out. Code transformation or

masquerading techniques have a high complexity, which is reflected in code sizes. They are also not applicable for every service, as many vulnerabilities of services supporting scripting are exploited, which provide a convenient, worm-size reducing facility (e.g., Visual Basic Scripting Language). At these levels masquerading is nearly impossible. Therefore code for such transformation techniques is not included in worms, as they do not improve the success of them, but are only unnecessary ballast. Therefore the same payload will be observable when a worm spreads.

The definition of trails has been used to reduce the search-space as well as to take causality into account. When a worm spreads, he infects a host, which also starts to show the worm search and spreading behavior. If a host has not been infected yet, then he does not show this behavior. Trails take into account where information might have come from, and therefore can exclude connections which are unrelated.

Regular communication patterns on the other hand do not show the behavior, with only a few exceptions, that exactly the same payload is sent from one host to another, the latter starting to send the same payload to other hosts. This only happens when some IP-packet is routed through the Internet and when the destination host has not been reached yet. Notice that during routing a packet is sent from one network to the other — this means that the same IP-datagram is only visible once in a network. Worms on the other hand try to infect all hosts, which are usually connected only to one network interface card. This causes that attack data is first sent to a host, and using the same network interface card the data is sent to another host. Therefore the data comprising the worm is sent through the same network at least twice when the worm tries to spread again.

To support our hypothesis we made an analysis of the DARPA'99 test data, which contains the network traffic of ten days. The trails of individual nodes have been generated from the network traffic and then within the connections of the trails recurring patterns have been searched for. The same test has been made with the traffic that has been collected from our department's network. Table 4.9 shows the results of this analysis, where we looked at repetitions of the same payload in the trails of the individual nodes. To make an analysis more efficient, we put several constraints on the connections which all had to be fulfilled to consider them as containing the same payload.

- **Minimum size.** For each connection we required that the payload of this connection has to be larger than a certain amount of bytes. This is motivated by the fact that available exploits consist of several hundred bytes, making it impossible to create worms with the whole propagation mechanism which only consist of only 32 (which we chose) or less bytes. Therefore this constraint does not affect the analysis, but can help with accelerating it.

- **The same destination ports have to be used.** Before the payload of two connections of a trail are compared, the destination ports of these connections have to be checked. When they are not equal, it is clear that they target different services. As each service requires customized, different exploits, it is unnecessary to compare the payloads of connections going to different services. Services that have been moved

to a different port can be ignored, as worms rely on the fact that services are usually bound to their standard ports.

- **Only meaningful packets are considered.** Connections that contain a payload consisting of a bulk of equal bytes are not meaningful and cannot be used to represent a worm.

- **The first *z* bytes of connection have to be equal.** We require that exactly the same payload has to be in the connections, i.e., the payload consists of the same bytes. As it is impossible to keep all connections with the full payload in memory, only the first *z* bytes of connections are taken into account. On the other hand this number does not have to be too low. In this case the similarities of protocol headers could lead to a detection of worms where actually there are none, as only the protocol headers or commands such as GET index.html are matched. For the analysis we chose *z* to be 512, which allowed to keep the consumed memory in an acceptable domain which ruling out similarities caused by protocol headers or commands.

Day	Size (bytes)	Nr. of observed payload repetitions in trails
Week 4, Monday	216724852	0
Week 4, Tuesday	301682860	0
Week 4, Wednesday	319141540	0
Week 4, Thursday	399619424	0
Week 4, Friday	262141472	0
Week 5, Monday	344257810	0
Week 5, Tuesday	419184640	0
Week 5, Wednesday	347930624	0
Week 5, Thursday	628236288	0
Week 5, Friday	969940992	0
Institute's traffic, 24h	278943824	0

Table 4.9: Analysis of Payload Repetitions Within Trails of Captured Network Traffic

To show that available worms show the behavior that has been stated by the hypothesis, we set up a testbed with a three hosts hat were running several vulnerable services. We launched `Nimda`, Code Red and Code Red II and captured the traffic. As soon as a service had been exploited, the worms started their activity. They opened many connections to other hosts and scanned for other vulnerable services. The target of the connections were in the local area network, but also random Internet hosts. As expected, the connection patterns that could be observed within the trails showed the same bytes for the same services (see Figure 4.10).

Regarding these results it is clear that an abstract signature for identifying worms has been found. On the one hand normal traffic did not contain any repetitions within the

Duration	Nr. of observed payload repetitions in trails
1 minute	17

Table 4.10: Analysis of Payload Repetitions in Worm Attacked Networks

payload of trails. On the other hand connections with the same payload could be observed in trails of network traffic containing worm activity.

A system can now be constructed that implements this abstract signature. Before this is shown, an extra requirement is introduced. As worm detection is only the first step in preventing security breaches, *automatic response* mechanisms are required to prevent a further spreading of the worm.

4.4.4 Response Mechanism Requirements

For the response mechanisms we had three requirements.

- automatic responses in near real-time The response mechanisms must be triggered as soon as signs for a spreading worm are spotted. In order to be efficient, they have to be executed at a speed comparable to the speed of the worm propagation itself.
- prevent infected hosts from infecting other hosts
- prevent non-infected hosts from being infected

In this chapter the automatic responses that are directed to stop worms from spreading is limited to firewall reconfiguration, as this are the most powerful method. Nevertheless, other response mechanisms might also be suitable, such as e.g., shutting down vulnerable services.

In general, the countermeasures serve two different purposes. The first is to prevent infected hosts from infecting others. Hosts that are identified to be infected insert rules into their firewalls that prevent outgoing traffic to the identified vulnerable services. This helps to prevent the worm from flooding the network with reconnaissance packets when it searches for new victims. The mechanism also helps to protect other hosts that are not directly covered by our proposed approach (e.g., hosts on the Internet). This is reasonable, because when the source for further infections can be isolated immediately, the rest of the hosts remain unaffected.

As an attacker could have included mechanisms into the malicious worm code to circumvent such protection approaches, the second requirement demands that non-infected hosts are protected from being infected. When the worm has been faster than the response mechanism on the infected host itself, the majority of hosts could, for example, be saved by disabling access to the vulnerable service. If access to services is not granted to infected hosts, the worm cannot spread anymore.

4.4.5 Basic Model

For our model, we assume that each node (host) participating in the network has installed a personal firewall which is configurable at runtime. The individual hosts can run arbitrary operating systems as long as firewall reconfiguration is possible. One host per local network segment (i.e., broadcast segment) runs a monitoring tool that puts the network interface into promiscuous mode and sniffs the TCP traffic. The monitoring station maintains a history of all recent connections and attempts to find recurring data in trail payloads, as shown in Figure 4.15. Notice that interesting patterns are not provided a-priori by external means — instead they are extracted from the monitored connections.

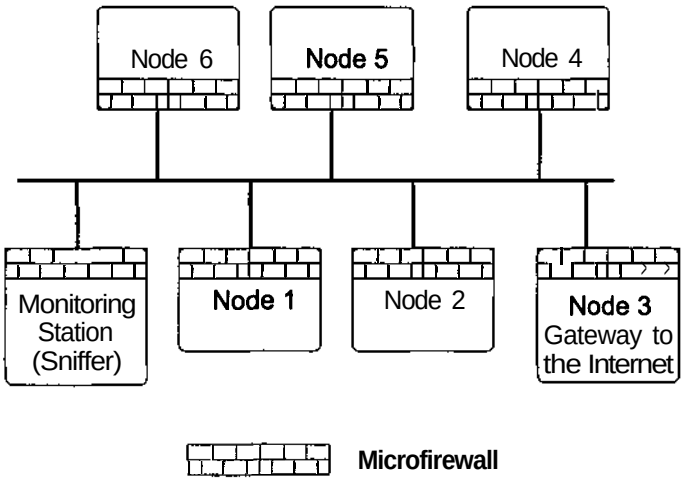


Figure 4.14: Worm Protected Network

While connections are monitored, the connection profile is built. After the behavior of a worm has been identified (packet payloads within the trail of a node are the same), the spreading mechanisms that the worm uses are identified too. Broadcasts from the monitoring station are used to disseminate the characteristics of a detected worm to the individual nodes of the network. At all hosts, the local firewalls are reconfigured to block the ports utilized by the worm's spreading mechanisms. This step is necessary to prevent the worm from infecting new hosts.

Even when not all possible ways have been identified that the worm can utilize to spread, the threat has been vastly reduced. When the worm attempts to use an alternate way for spreading (trying an exploit against a service that has not been seen in the node's trail), it will be detected very soon and the firewalls are updated accordingly. Already identified attacks against services are no longer effective as they are blocked at the firewall.

4.4.6 Efficient Worm Pattern Identification

As mentioned above, a central analyzer collects the connection data of all nodes in a local area network segment. Two data structures, which are kept for every monitored node on the

monitoring station, are utilized for an efficient worm pattern identification — a *connection history* and a *connection trap list.* The connection history of a node N is denoted as CH_N, while the connection trap list of the same node is denoted as CTL_N.

The connection history contains all connection tuples that have been opened **to** that host. This includes data extracted from the first few TCP segments (i.e., the first *z* bytes). An *ignore list,* which is also provided on a per-host-base, allows the specification of events which should not be inserted into the connection history. This a-priori knowledge helps to reduce the storage overhead by focusing on interesting occurrences. With the *ignore list* many chains can be eliminated that would otherwise have to be considered as parts of worms, which would raise the matching effort.

The connection trap list summarizes the information of a node's trail. Its elements are lists that represent the chains that end at this node (i.e., one element for each chain in this node's trail). This obviously includes connections that do not directly end at that node, but are in chains which pass through this node. Note that the trail construction can be made incrementally, as only merge operations are necessary. When a connection to a certain node is made from a source host to a destination host, the destination host's trail only has to be merged with a modified version of the source host's trail. The source host's trail only has to be extended by the just performed connection and merged with the existing trail. Figure 4.15 shows a connection *q* that is made from node A to node B.

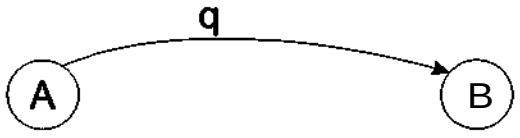


Figure 4.15: A Sample Connection Between Two Nodes A and B

The algorithm for constructing the connection history and the connection trap list of a node is shown below. Assume that a new connection is opened from host A to host B.

Algorithm:

1. Only if the connection history and the connection trap list of host A are empty, add the connection q to the connection history of host B.
2. If at least one of the connection history or the connection trap list of host A is not empty, let the connection history of host B be unchanged.
3. Construct the set l by appending connection q to all chains of the connection trap list of host A.
4. Construct the set m by appending connection q to all chains of the connection history of host A.
5. Update the connection trap list of B by adding l and m setwise

Formally the relation between the CH_A and CH_B is shown in Equation 4.4, the relation between CTL_A and CTL_B is shown in Equation 4.5.

$$CH_B = \begin{cases} CH_B \cup \{q\} & \text{iff } CH_A = 0 \text{ and } CTL_A = 0 \\ CH_B & \text{in any other case} \end{cases} \tag{4.4}$$

$$CTL_B = CTL_B \cup \{l\} \cup \{m\} \quad \begin{matrix} l = append(t, q), t \in CTL_A \\ m = append(s, q), s \in CH_A \end{matrix} \tag{4.5}$$

The function *append* adds an element at the tail of a chain. l represents all the connections in the trail of the source host that have been extended by the connection q. m represents the chains that have been created by merging the connections from CH_A and the new connection q. As can be seen, connections are inserted into the connection history of node B only, if the connection history and the connection trap list of node A is empty. Otherwise it is possible to create a chain by extending the elements of node A and directly add them to the connection trap list of node B.

When an outgoing connection is detected that matches an entry of any element of a list of this connection trap list (i.e., a connection with a destination port and a content that is exactly similar to an element of the trap list), a part of a potential worm pattern has been found. The propagation mechanisms the worm used for spreading can be directly read out from the chain in which the recurrence of the payload occurred. The worm at least uses the mechanisms for propagations that have been used between the two elements in the chain. Imagine that a chain consists of connections using the services HTTP, FTP and HTTP and the in payload recurrence was detected when the HTTP service was used. Then the worm uses the HTTP and FTP services for spreading. This is because the worm needed to get from the destination host of the first connection to the source host of the third connection. As an FTP connection is the only connection between them it is only plausible to include FTP into the mechanisms that are used for spreading.

Recall the connection graph from Figure 4.13. Table 4.11 shows the individual connection histories and the connection trap lists for every node. If nodes are not listed, their respecting connection histories and connection trap lists are empty. Connections 3 and 6 are filtered out, as their destinations could not be reached.

The result of this process is that a worm has been identified that uses several ways of spreading. The worm at least exploits the services HTTP, SMPT and DNS, as the same payload could be observed in the chain 1-5-7-8.

4.4.7 Implementation

We have implemented a prototype of the system that realizes the above described approach. A single node runs a network sniffer in promiscuous mode (with full TCP stream reassembly) together with the above explained model which is realized as a plug-in. In combination with a dynamic firewall configuration daemon at all hosts this is enough to perform the desired tasks. While the sniffer collects the available network traffic (TCP and UDP) and detects worm signatures the daemons configure the firewalls at the other hosts accordingly.

Connection	Node	Connection History	Connection Trap List
1	4	{1}	O
2	4	{1,2}	O
4	3	{4}	O
	4	{1,2}	O
5	3	{4}	O
	4	{1,2}	O
	5	{5}	{1-5,2-5 }
7	3	{4,7}	{1-5-7,2-5-7 }
	4	{1,2}	O
	5	{5}	{1-5,2-5 }
8	2	O	{4-8, 1-5-7-8, 2-5-7-8}
	3	{4,7}	{1-5-7,2-5-7 }
	4	{1,2}	O
	5	{5}	{1-5,2-5 }

Table 4.11: Connection History and Connection Trap Lists for the Individual Nodes

Our prototype has been implemented in C on a Linux 2.4.18 (SuSE 8.1) system, as a Snort plug-in. Parameters for the model (such as the time to keep connection data, the amount of data that should be kept for each connection etc.) are configurable via Snort's configuration file, which is read at startup time. Because memory is a limited resource, only a very limited amount of connection data can be kept in memory. Therefore, the connection history only stores connection data of a certain, user definable period of time. This can be justified because the strength of worms is their ability to spread very quickly over whole networks. Due to the fact that only a few bytes have to be stored for a single connection, the likelihood that the system's memory will be insufficient is very low.

The daemon uses the iptables module to insert into and remove firewall rules from the system. These firewall rules can be configured such that they only remain active for a limited amount of time. This is motivated by the fact that hosts that are infected with the worm also have a firewall that has closed the outgoing ports. When the worm activity has stopped (because the personal firewalls have been reconfigured according to the worm patterns) the ports used by the services can be made available again. But this is only done if it is sure that the worm did not attack the firewall, i.e., shut down the firewall and try to spread again.

The communication channel that is used to broadcast reconfiguration and worm property information over the network is cryptographically protected. This is done by calculating an MD5 sum of the payload that is sent out from the monitoring station to the individual client daemons and encrypting it with its secret key. The public key of the monitoring station is present at all hosts of the network. Using this public key, the authenticity of the message can be determined by the daemon (that obviously only accepts valid and authentic messages).

4.4.8 Experimental Data

The network for performing experiments consisted of three hosts running different operating systems. Different services (e.g., HTTP, DNS, sshd) as well as personal firewalls have been installed at each. A single machine was running our prototype implementation with the secure infrastructure as explained above. For our tests we set the timeout for the connection history and the connection trap list to 1800s (half an hour) and we took into account the first 512 bytes of a connection. We used the first 512 bytes of each connection to prevent false positives. If this number is set too low, many connections would be identified as containing worms, as their payload is the same. This is caused by the fact that most services use a protocol header which is very likely to be the same (such as GET in HTTP requests). We made three experiments with the gained prototype.

The first experiment we aimed at testing the principal functionality of the prototype. The behavior of different worms has been simulated with `tcpreplay` [83], including connections to non-existing hosts or services. In all seven cases, our system has been able to reliably detect the worm pattern. Especially the inclusion of the payload has been valuable, as it allowed the monitoring station to identify similar connections (e.g., identical ports) with different payloads as different without causing the system to detect this case as an alarm (reducing the false positive rate). As an optimization, we inserted known connections occurring in our network into the ignore-list which then reduced the amount of data that had to be processed.

The second experiment is actual a repetition of the experiment that has been performed for verifying the hypothesis. We launched several worms against a local area network of three hosts that did not have a connection to the Internet. The monitoring station watched the traffic that was exchanged between the nodes of the network and detected the worms — sending out directives for the individual personal firewalls. With this scenario the system followed the network activity easily on a heavy loaded network, as only a few bytes per connection have to be taken into account. The average message length that has been broadcast by the monitoring station was 81 bytes. The calculations of MD5 sums of the request payloads lasted 0.002ms on average, while the encryption of the MD5 sums lasted 0.085ms. These measurements were taken on an Intel Pentium III with 550 MHz and 512 MB of RAM.

The modification of the dynamic firewall rules starting from the point when the monitoring station initiated the action to the activation of the rule lasted 23ms on average. As the propagation time of worms is in the same range, it is justified that the system can protect all hosts of a network with appropriate speed once a worm has been detected. A worm always tries to infect one host, but once its propagation mechanisms have been identified, the whole network is protected. We noticed that the protection mechanism prevented vulnerable services from being infected by the worm. Clearly not all of the services remained uncompromised, as it was necessary first to detect the worm. Usually the same service within a network had been infected two times (at different machines). The monitoring station had then identified the worm and the propagation mechanisms and broadcast out the relevant data to all the nodes of the local area network, which immediately changed

their firewall rules. If one considers a network with many hosts, this is a good ratio, as most of the hosts remain uncompromised.

The third experiment should clarify whether the performance of the model is sufficient during regular conditions. Therefore we used our prototype to process the DARPA'99 evaluation data again and measured the required time.

Day	Time (s)
Week 4, Monday	218.13196
Week 4, Tuesday	304.48202
Week 4, Wednesday	372.30696
Week 4, Thursday	503.55325
Week 4, Friday	262.81705
Week 5, Monday	283.23756
Week 5, Tuesday	326.13771
Week 5, Wednesday	785.712154
Week 5, Thursday	1005.81205
Week 5, Friday	2299.104822

Table 4.12: Analysis of Required Time to Process the Captured Network Traffic

The results show that 24 hour traffic could easily be processed within a short time, making clear that the detection algorithm is efficient. As long as the TCP reassembling routines can handle the traffic load, the worm detection routes can do so, too.

4.5 Summary and Conclusions

In this chapter the concept of abstract signatures has been introduced. Classical signature-based IDS require a set of signatures that are used for exact pattern matching within events, thereby detecting intrusion relevant events. Signature-based IDS require a constant update of the signature database to be useful and cannot be used for detecting unknown intrusions, which is their major shortcoming. In contrast to classical signature-based systems abstract signatures do not require a set of signatures for intrusion detection. Instead of specifying the instances of an attack-class, the similarities of the attack-classes are expressed. Usually a model of the attacks is created that implements a detector for exactly these similarities. Abstract signature sensors detect attacks by applying the model to the individual events, determining whether events contain instances of attacks. As a consequence of modeling similarities of attacks, an abstract signature can detect attacks that belong to the same class the model has been created for. This clearly results in the fact that abstract signatures are update free in the sense that no new signature instances have to be inserted into some signature database, as the abstract signature represents all signatures of this class. This is a nice property of abstract signatures, as they allow a detection of previously unknown attacks, which is the major shortcoming of existing signature-based systems.

In contrast to anomaly-based systems abstract signatures do not build up a profile and then monitor some behavior for deviations from this profile. Instead, the model captures the similarities of the attack class and then exactly watches the generated events for this similarity. The problems of anomaly-based systems, the constant necessity of training systems and their high false positive rates are not a matter for abstract signatures. Training abstract signatures is not necessary, as a human being already encoded the mechanisms that are used for identifying whole classes of attacks. Before he could do this, he first had to analyze the attack-class and find a hypothesis which he could verify experimentally or prove formally. After having done this, he implemented a sensor that contains the gained knowledge about similarities of an attack-class. The sensor therefore can identify all attacks belonging to the same class. A basic requirement for abstract signatures was that they have to have a very low false positive rate to make them more useful than anomaly-based systems. This requirement has been met during the development phase of the analyzer of the attack. After the analyzer has built a hypothesis of a similarity, he has to verify or prove this similarity somehow, which might happen experimentally by analyzing a large amount of data or by proving it formally. This process is a two step procedure. At first it has to be shown that all existing attacks of this class have this similarity. The second step has to show that this similarity does not occur in normal operations. Once this is shown, it is clear that the resulting abstract signature will have a very low false positive rate (which actually should be zero), as the similarity only has been observed in attacks, but not in normal event data.

Abstract signatures have another advantage, as they can work much more efficiently than normal signature-based systems. Usually abstract signatures represent thousands or even millions of signatures. This means that a common signature-based ID system would have to use this amount of signatures in its signature database to be able to detect the same amount of attacks. But as the number of used signatures is a limiting factor in intrusion detection (see Chapter 3), the performance of such a system would be very bad. Abstract signatures on the other hand do not need to generate all the signature instances for being able to detect the attacks — they are just able to detect the attacks, which is the effective goal of IDS.

We have presented two different sensors that implement abstract signatures — one detects buffer overflow exploits while the other detects worms.

The sensor for detecting abstract signatures is based on the fact that buffer overflow exploits contain some special code that makes the actual exploit work on different machines. This code, the sledge, just leads to the exploit, nevertheless it is executable and cannot be encoded or encrypted. We developed some metrics for detecting the sledges by making use of their executability. We set up the hypothesis that normal requests do not contain long executable instruction chains, but buffer overflow attacks do, which we proved by exhaustive experimental analysis for the services HTTP, DNS and FTP. We created a prototype sensor of this abstract signature which has been implemented at an Apache module that checks incoming requests. If a request is caught that has an unusual long MEL, the request is disallowed to be processed, preventing it from being processed by possibly vulnerable Apache functions. The performance measurements gained through an

analysis using WebSTONE showed, that the abstract signature works very efficiently. An obvious shortcoming of the proposed approach is that it cannot detect exploits that utilize methods to avoid executable sledges. Vulnerable services that include debugging routines that output information which might be used to calculate the exact stack address can be exploited by hackers. If the attacker causes the service to execute the debugging output and calculates the *exact* stack address (info-leak attack), he can create buffer overflows that do not include executable sledges. But as only very few services include these debugging routines our approach for detecting buffer overflows is still useful.

We have also presented the design and implementation of a system that monitors a network for connection patterns that represent spreading worms. Like in the buffer overflow detection case, the signatures that specify a worm are not provided a-priori. The system itself identifies the characteristics of the worm by deriving the relevant information by searching for similarities of connections in the same trail.

We set up the hypothesis that worms use the same payload for attacking uninfected hosts, which was supported by the fact that code transformation techniques are code intensive and cannot be applied for every attack. The observation that worms such as Nimda use several services for spreading made the whole situation a little bit complicated, as it was not sufficient to compare outgoing connections of a host to the incoming. We introduced the notions of connection chains and a so-called *trail* of a node, which allowed us to formulate a hypothesis. Experimentally we proved that worms show similar connections within the trails of infected nodes while normal traffic does not contain these similar connection patterns. Our prototype implementation is realized as a plug-in for Snort-1.9.1 that detects worms and securely broadcasts the identified worm pattern to all nodes of the local network. These then reconfigure their firewalls and prevent incoming connections to the services a worm used for spreading. Additionally the infected hosts close down outgoing connections to services the worm uses for spreading. We showed that our prototype implementation is efficient and is able to launch the automatic response mechanisms within a very short time. This enables the system to prevent worms from spreading, ensuring the security of the individual nodes. Additionally this mechanism is effective in preventing the massive denial of service attacks that are caused by worms, which scan for new vulnerable hosts. Worms are even a threat for machines that do not run any vulnerable services, as the scanning activity of worms usually consumes all the available bandwidth — this makes it impossible to run a business without interference, as the current Internet cannot provide any bandwidth guarantees. As our system also prevents outgoing connections, these search activities are not allowed once the worm behavior has been identified. Therefore our system can help in preventing the DOS attack caused by the scanning activity of worms once it is deployed widely.

Chapter 5

Response Impact Evaluation

> We have to distrust each other. It's our only defense against betrayal.
>
> Tennessee Williams (1911 - 1983)

In this chapter, we present a network model [82] to evaluate the effects of intrusion response mechanisms on the operation of network services to meet the goal of keeping the usability of a system as high as possible. This work tackles the problem of automatically setting responses which degrade the usability of a system as shown in Chapter 2. Static mapping tables are not flexible enough for deciding how to react in a certain situation, as they work on oversimplified information and cannot represent the current state of the network. We present a network model that provides a method for specifying the mission of a network and its individual components as well as the dependencies between different entities to capture the consequences of responses more accurately. Based on this model, an evaluation function can estimate the impact of various responses and select the one with the expected minimal negative consequences. This model and the evaluation function can be seen as a subsystem of an automatic intrusion response system to make the use of intrusion response safe.

The next two sections discuss the requirements for the subsystem and the network model itself. Section 5.3 describes the evaluation function to calculate the effects of responses, while Section 5.4 presents the implementation. Then we provide experimental results that have been obtained from our prototype implementation. Finally we evaluate our approach.

5.1 Model Requirements

This section elaborates on the requirements that we have identified for our network model to be able to accurately calculate the effects of responses. The basic idea is to determine whether information can flow between the individual entities of the network. The main requirement is to be able to capture the current state of a network with respect to the network's mission. To meet this requirement, the following requirements have been identified.

- **Flexibility.** The model has to be able to cope with different network topologies and must be able to express the dependencies among resources themselves and between resources and users. As there should be no artificial restrictions, our model should not have to rely on simple mapping tables for calculating response effects. Instead, the actual situation of the network needs to be reflected in the model to be able to determine the effects of responses accurately. Not all resources or users have the same relevance for the operation of the network — this fact has to be reflected in the model. A resource that is only utilized by low priority entities is obviously less important than one used by a mission critical entity. The importance of resources can vary dynamically — even by the time of day.

- **Dynamicity.** The model has to be dynamic to be able to track changes in the environment (caused by response actions). A reconfiguration of the firewall has to be reflected in the model, as well as changes in the availability of services due to their (de)activation by a response. This is clearly a requirement, as calculating the effects of responses requires a knowledge of the current situation, which often changes by launching responses.

- **Efficiency.** In order to be useful, the model has to deliver results quickly. IRS have to respond fast to keep the time window of vulnerability small, therefore calculating the effects of responses should be done in near-realtime. The model should make the design of an efficient evaluation function possible.

- **Ease of use.** In a large network, there are many dependencies between different entities. For making administration of our proposed system easy, not all of them should have to be entered explicitly by an administrator. The majority of dependencies have to be determined automatically by an analysis of transitive relations and the network topology. Only basic relationships at a high level (e.g., this host needs access to one of the DNS servers) should have to be specified.

  The model should be intuitive and comprehensive in the sense that it resembles the facts of the real world. A smooth integration of entities and their dependencies is necessary to achieve this goal.

- **Minimization of negative impact.** The model has to be capable of helping the IRS in determining which responses to use. In the case that more than one response action is available, the one with the least negative effects on the usability on the whole system should be chosen.

5.2 Network Model

The main means for self defense are reconfiguring the security policy and dis/enabling resources. Therefore our model has to be able to handle these changes. Although the model is applicable in general, we focus on firewall rules modification, enabling or disabling user

accounts and modifying the status of processes running on a host (i.e., restarting network services, terminating malicious programs).

The following section introduces our network model that is used to calculate the effects of response actions. First, the elements, which are included in our model, are identified. The basic elements, as explained in more detail below, are services provided by hosts (such as a WWW server), users of the network, the underlying communication infrastructure and the deployed security policy.

Then, we show the differences between direct and indirect dependencies between these entities. After that the algorithm that operates on the model to determine the effects of responses is explained.

5.2.1 Modeled Elements

Networks are complex structures that include many elements which are heavily related and dependent on each other. For our model, the following elements are relevant. Note that the notions system users and resources are together referred to as *entities*).

- Resources. Resources represent network services that are offered by servers to clients. They are the basic building blocks of our network model. As not all resources are stand-alone, some of them require other resources to a certain degree for providing their own service. Network services such as DNS, NFS, NIS, HTTP or FTP are provided to its consumers by processes that listen on a predefined network port. Consumers send messages to these ports that contain the requests the clients need to be answered. The requests are processed by the server and reply messages are sent back to the originators of queries. It is important to mention that in our model only resources that are used by other entities have to be included — processes running at a host without providing services to external entities do not have to be considered to be resources.

- System users. Users have to perform their tasks by utilizing the provided resources, therefore they have to be part of the model as well. Users can assign different levels of importance to resources, as they depend on these with different degrees.

- Network topology. The network topology has an important role for the evaluation process because it determines the communication framework utilized between different resources. The path that is used for delivering messages between two hosts is defined by the available communication lines and the used routes.

- Security policy. The installed security policy effect the availability of resources/services of the protected network. Dependencies between two resources located in the same subnet are e.g., not affected by response effects that modify firewall rules. In the case that the communication path between entities leads through a firewall, the deployment of new firewall-rules obviously influences the availability of that resource.

5.2.2 Entity Dependencies

This section explains the dependency relationship between two entities in detail. The differences between *direct* and *indirect* dependencies are shown.

Definition:

> An entity *A*, which needs a service that is provided by another entity *B* to be fully operational, is called *dependent* on entity *B*. The relation between these two entities is called a *dependency*.

Among the different entities which are distributed over the hosts of a network, there are many dependency relationships. While some entities do not need other ones for being fully operational (stand alone machine without network connections for running editors) typical machines require access to mail-servers to let users send and receive e-mail, to DNS servers to allow DNS name resolution or to an HTTP server for accessing web pages.

An entity is considered to be *available* for a dependent one if:

1. communication between both is possible, which means
   (a) there is a route provided by the underlying network topology between both and
   (b) all hosts on the route permit the traffic that is exchanged between the two entities.
2. the entity providing the service is functional (i.e., the process providing the service is running).

Figure 5.1 shows the dependencies among two users Anne and Customer, two HTTP and DNS servers as well as an NFS server. The entities are expressed as annotated boxes while the dependency relationships are expressed as arrows.

Definition:

> The dependency between two entities may be *direct* or *indirect*.

A *direct dependency* is a dependency that is given to the model manually. These are the dependencies of entities at the very basic level. An example would be a user that uses the DNS service to resolve DNS names (e.g., user Anne in Figure 5.1), therefore she needs access to one of the two DNS servers.

As described above, we consider the network topology and the security policy as part of our network model. While the network topology is the glue between the resources by providing communication paths, the security policy (especially firewall rules) can be viewed as a method for imposing constraints on these paths by (dis-)allowing certain traffic. The network topology and the security policy introduce new artificial, *indirect* dependencies between entities and their needed resources. This is caused by the fact that information exchange has to take place over several hosts (e.g., routers) and must be allowed by all

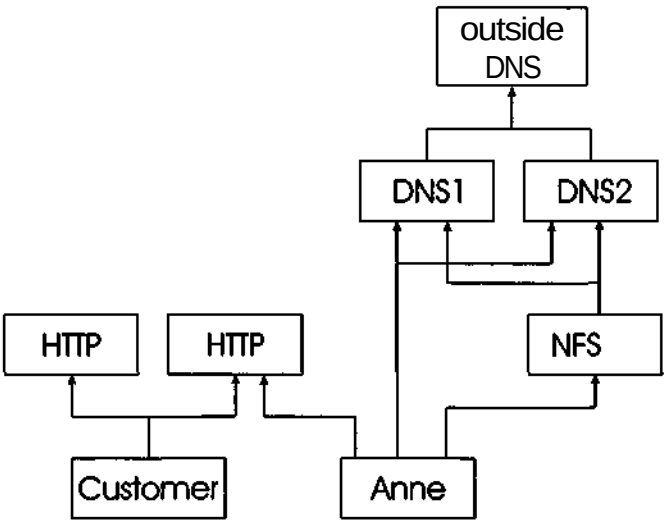


Figure 5.1: Entity Dependencies

security policies that are in effect on the communication path (i.e., host personal firewalls, router firewalls). These artificial dependencies are called *indirect dependencies.* Indirect dependencies do not have to be specified, because they are determined automatically by analyzing the network topology (which is encoded in routing tables) as well as the security policy (which is encoded in the firewall rules scripts). Indirect dependencies can be seen as a precondition for fulfilling their corresponding direct dependency. All the indirect dependencies that are imposed by nodes being on the path between two depending entities have to be fulfilled to fulfill the direct dependency that created the indirect dependency. In the case that a direct dependency induces an indirect dependency that is not fulfilled (e.g., the packet is filtered at a certain node on the communication path), then the direct dependency can never be fulfilled, too. Indirect dependencies would be immediately introduced in the example shown in Figure 5.1 if the DNS servers are located in a different subnet than Anne or if personal firewalls are installed on any of the just mentioned hosts.

Indirect dependencies can be determined when the direct dependencies, the network topology as well as the security policy are combined. They have to be generated once, when the model is initialized and then only have to be checked. This is clearly possible as long as one assumes that the routing infrastructure will remain the same — as soon as the routing tables of hosts change, the indirect dependencies have to be recalculated, as other communication paths might be used during information exchange.

The example in Figure 5.2 shows a network that consists of four subnets and the external Internet (in the top left corner). The direct dependencies are identical to those shown in Figure 5.1. However, note the indirect dependencies between the gateways that connect the different subnets. In this figure, firewall rules are omitted for the sake of simplicity. Direct dependencies are not displayed anymore, instead the indirect dependencies that have been generated by combining it with the network topology and the security policy are displayed. They lead from one entity to the entity it depends on and show the path that is taken for

information exchange. The routing information is visualized through dashed arrows. The dependencies of the DNS servers to the external DNS server have been omitted for simplicity.

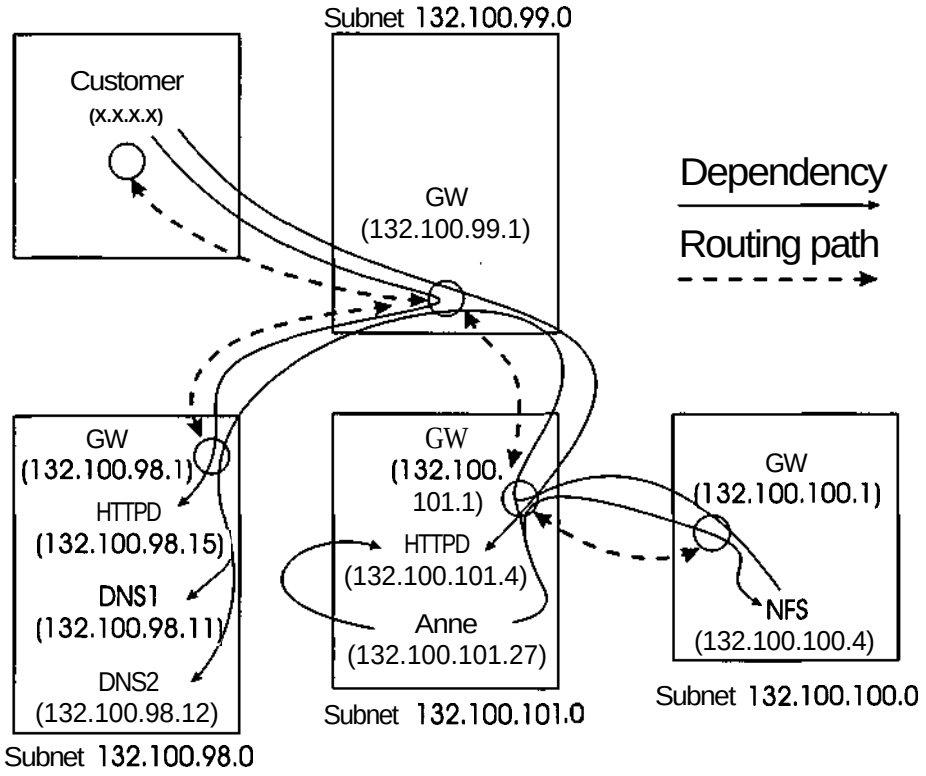


Figure 5.2: Topology and Entity Dependencies

5.3 Impact Evaluation

After the dependencies between entities have been elaborated, an approach for evaluating the impact on the mission of a network is presented.

Definition:

> A *response action* is a set of operations that can be utilized to avert a certain, identified threat. The basic operations, called *response items*, are basic steps such as changing the security policy (installing or removing firewall rules), killing and restarting processes or user account en-/disabling.

Response actions are initiated by the IRS in response to an intrusion that has been detected by an IDS. Because a number of different response actions might achieve the desired result, it is the task of the IRS to choose the one with the least impact. The determination of the impact (or effect) of response actions is done with help of the current network model by an *impact evaluation function*, which calculates the degradation of the

usability of individual entities and also of the whole network. The set of response actions that have already been applied clearly have to be taken into account to be able to perform an exact impact calculation. The response actions that have lead to the current state of the network are called *response configuration* and serve as an input for the impact evaluation function.

A single response action can affect entities either directly or indirectly. A direct effect is witnessed when a needed service becomes unavailable (e.g., due to stopping of services or disabling of user accounts). An indirect effect is experienced when the direct effect on one entity reduces the service that this entity can provide to another one, thereby affecting entities that are unrelated at first glance but which are indirectly dependent.

If a response action hits an entity, it will not be able to perform its task with the same quality or speed as before. The *degree* of a dependency describes in how far the operation of an entity is affected if the resource, which it depends on, is no longer available.

The introduction of a degree of dependency can be best motivated by the following example. Consider a user that uses his machine mainly to access hosts from the Internet. In our network model, the entity representing this user will depend a lot on the availability of the DNS and HTTP servers (the dependencies will have high degrees), but not on the NFS server. On the other hand, a user editing files on the remote NFS machine, such as user Anne, will mainly need this service to accomplish the work. Here the dependency degrees of the dependency with the NFS server will be much larger than the ones with the HTTP and DNS servers.

An entity will usually depend on several resources in the network. These relationships do not necessarily have to be trivial. For some entities, it is sufficient to have access to at least one of a set of (similar) services (called 'OR-dependency') while others need access to all of them (called 'AND-dependency'). Our model is capable of expressing both types of relationship as well as combinations of them using *dependency trees.* Dependency trees are binary trees, that reflect the other entities an entity depends on. The leave nodes are the other entities it depends on, while the intermediate nodes of the tree determine the exact relations. The edges between the individual nodes of the dependency tree are annotated with the dependency degree.

The following example describes the direct dependencies of user Anne, who requires mainly access to the NFS server and to one of the two domain name servers DNS1 and DNS2. Additionally she sometimes requires access to the HTTP server. These relationships can be denoted in a *dependency tree* as shown in Figure 5.3.

The *capability* $c(r)$ of an entity r is a value ranging from 0.0 to 1.0 and describes in how far a resource can perform its work given the current response configuration, compared to the situation where all needed resources are available. The calculation of this value is based on the underlying network model given its current state (with services that might have already been disabled or security policies might have been changed) and the entity's dependency tree. When the capability value is determined for an entity, the communication paths to all the resources that it depends on are examined. This allows the evaluation function to take the current routing and packet filter (firewall) rules into account. When specifying a dependency tree for a certain entity, one must make sure that the capability

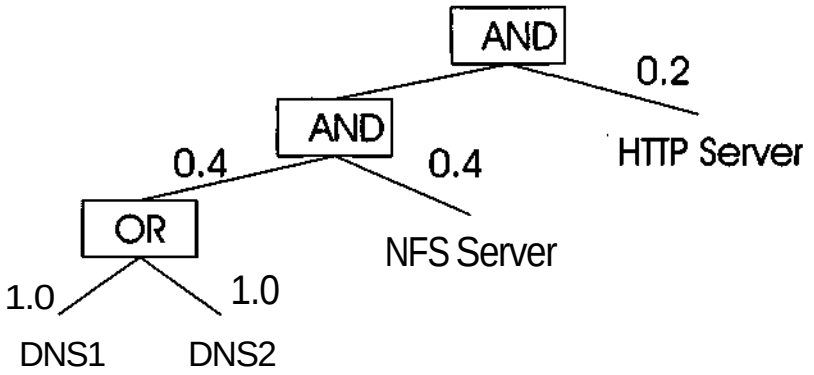


Figure 5.3: Dependency Tree

of the entity is 1.0 when all resources are available.

5.3.1 Capability Calculation

The following paragraphs explain how a capability value is determined for an entity. Two basic cases have to be distinguished for capability calculation.

1. If the entity does not depend on other entities only the current condition of the entity determines its capability.

   - if the entity provides service, the capability is set to 1.0.
   - if the entity does not provide service, the capability is set to 0.0.

2. If the entity depends on other entities, a recursive algorithm that performs a depth-first search on the dependency tree is utilized to determine its capability. The types of the nodes of the dependency tree determine which formulas are used to aggregate the capability values obtained from the subtrees below. The intermediate nodes of the graph can be either of the type AND or OR, while the leaves represent entities. $func(left)$ and $func(right)$ denote the capability of the left/right link of a node, multiplied with the dependency degree. c describes the capability value that is derived for the intermediate node. The items of the following list denote the three different node types.

   - **Leave node — Entity.** In this case, the value c for the leaf node is set to the current capability of this entity.
   - **Intermediate node — OR.** $c = max(func(left), func(right))$
   - **Intermediate node — AND.** $c — func(left) + func(right)$

To make this evaluation process efficient, no cyclic dependencies may be present among all the resource dependencies. This clearly makes it possible to construct a fixed evaluation

order in which each entity has to be evaluated only once. The order can simply be generated by expanding all dependency trees. During this operation, all leaves in a dependency trees are substituted with their dependency trees. No cyclic dependencies are allowed, therefore the trees have a bounded size. The evaluation order is then determined by the trees themselves. The elements at the bottom of the trees have to be evaluated first, and then the ones one level up in the tree can be evaluated.

Definition:

> The *capability reduction* $cr(r)$ of a resource r is the value 1 — c(r). The *penalty cost* for an entity is a value representing the cost when this entity becomes unavailable. The penalty-cost $p(r)$ of a resource can be calculated with Equation 5.1 below.

$$p(r) = cr(r) * penalty \tag{5.1}$$

where the penalty is a user-defined constant that reflects the importance of an entity. The constants can have arbitrary values, but have to be chosen in such a way that the relative importance among resources is reflected.

As an example, consider the penalty for the web server of an e-commerce site. Here, the penalty will be extremely high as it is necessary to have a running web server to stay in business. On the other hand, the penalty for the same service (web service) of a normal company will be usually lower. Downtimes of the webserver are clearly acceptable in that case, while development machines may not.

Definition:

> The *overall penalty cost* $pcost_{overall}$ is calculated from the penalty costs of the individual entities, as shown in Equation 5.2.

$$pcost_{overall} = \sum_{Vr} p(r) \tag{5.2}$$

5.3.2 Cost Optimization

Consider the situation where a threat or an intrusion is identified. There are often a variety of possibilities where and how a response action can be deployed. Nevertheless, the choice of the actual response item or response locations can have a tremendous impact on the usability of the whole system.

Response actions that effect the system's security in similar ways (i.e., that counter a certain threat) are called *alternatives.* Ideally, a response system can determine a number of adequate response actions which all provide the same level of security. In this case, the response action with the least impact should be chosen to keep the usability of the network as high as possible. Assume a situation in which a denial-of-service (DOS) attack against the HTTP server `132.100.101.4` in Figure 5.2 is detected, which has its origin in the

Internet. The response system might then decide to prevent outside traffic to this machine either at the gateway to the Internet or at the gateway located on the same subnet as the HTTP server.

Usually, choosing the best alternative is a difficult task. But by determining the impact (i.e., penalty cost) of a response action on all entities of the network (using our model and the evaluation function), the one with the lowest negative effect (i.e., the lowest penalty cost) can be selected. With our model we can perform two different types of calculations:

1. **Minimizing the penalty cost of new response actions.** This can be easily done when the IRS determines and presents appropriate alternatives to our system. Each alternative is simply added temporarily to the model, the model is evaluated and the overall penalty cost $pcost_{overall}$ is determined, and then the alternative is removed. The response action with the lowest penalty cost can then be chosen by the IRS, as shown in Figure 5.4. Obviously, the actually launched response actions have to be added permanently to the model (more precisely, to the response configuration) to keep it up-to-date with the actual state.

2. **Minimizing the overall penalty cost.** When the response with the least impact is chosen in every step (the local optimum), the overall response configuration might not be globally optimal (see the example in Section 5.3.4). Finding a globally optimal response configuration is not trivial and a number of previous actions might have to be 'rolled' back. All alternative combinations of response actions (stored in a so-called *response history)* have to be re-evaluated to find the scenario which has the least overall penalty cost. This is clearly a very costly operation, as the computational complexity for performing global optimization rises exponential to the number of already set response actions.

5.3.3 Model Language Grammar

Now the grammar that has been developed for specifying entities and their direct dependencies are presented. Each relevant entity has to be specified to include it in the response impact evaluation. To keep the grammar simple and intuitive, properties of entities have been introduced. These provide a facility for setting attributes to specific values, i.e., the cost of an entity being unavailable or only partly functional.

The grammar declares for each entity the header and the properties. Note that not all entities have external dependencies, therefore the depends part is optional.

5.3.4 Example

This section shows the network model of the simple example that has been introduced above in Figure 5.2 written in our grammar.

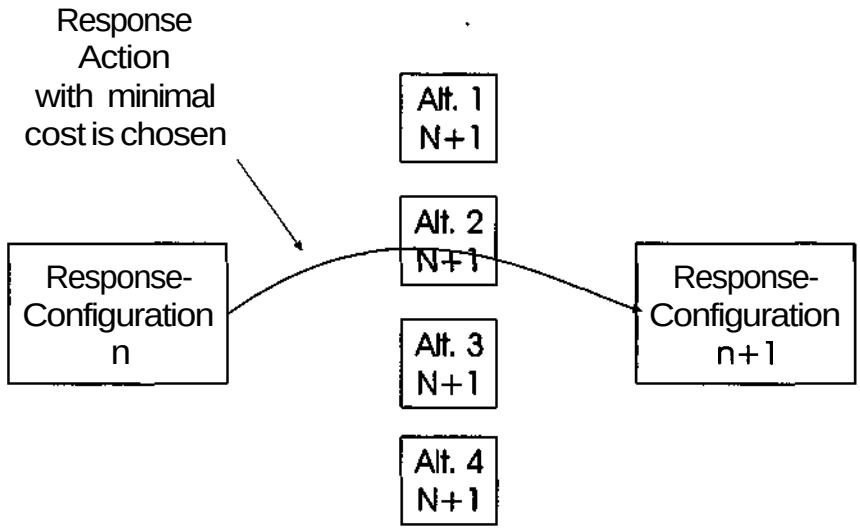


Figure 5.4: Response Configurations

After that, we discuss the performed calculations when different response actions are simulated and demonstrate that the order in which response actions are selected is crucial for the final result. Our example presents a situation where the selected responses, though locally optimal, do not lead to the best global result. It is shown that only an expensive global optimization which involves brute forcing all possible combinations can assert that a setup with a minimal global penalty cost is reached.

```
DNS is service at 132.100.98.11 53 udp,
  at 132.100.98.12 53 udp
   {processName="bind";};

HTTP is service at 132.100.98.15 80,
  at 132.100.101.4 80
   { processName="httpd"; };

NFS is service at 132.100.100.4 2049
  { processName="nfsd"; >;

anne is user at 132.100.100.27
   { cost=5000; }
  requires (
   (DNS at 132.100.98.11 53 udp or
    DNS at 132.100.98.12 53 udp) 0.4
    and
   (NFS at 132.100.100.4 2049 0.4
    and
   HTTP at 132.100.101.4 80 0.2 ));
```

```
customer is user at
  !132.100.0.0/255.255.0.0
  tcp { cost= 100000; }
  requires
   (HTTP at 132.100.101.4 80 1.0
    or HTTP at 132.100.98.15 80 1.0 );

// routing entries & fw rules
// subnet *.98
{
132.100.98.1 132.100.98.0 0.0.0.0
  255.255.255.0 eth1;
132.100.99.16 132.100.99.0 0.0.0.0
  255.255.255.0 eth0;
132.100.98.1 0.0.0.0 132.100.99.1
  255.255.255.0;
}
// subnet *.99
{
```

```
Entity          : <header> <properties> [ 'requires' <depends>] ';'
header          : entityName 'is' <type> <locations>
type            : ( 'SERVICE' | 'USER' )
locations       : 'at' (IPAddress/subnetmask I Hostname)
                  |'at' (IPAddress/subnetmask I Hostname) ',' <locations>
properties      : '{' (<property> ';')+ '}'
property        : Attribute '=' Value
depends         : ( <compounddepend>['or'])+
compounddepend  : '(' resourceName <location> [<degree>]
                  ['and' <depends>]+ ')'
degree          : <number>
```

Figure 5.5: Grammar for Specifying Direct Dependencies Among Entities

```
132.100.99.1  132.100.99.0   0.0.0.0
  255.255.255.0 eth0;
132.100.99.1 0.0.0.0 132.101.27.1
  255.255.255.0;
132.100.98.1  132.100.98.0  0.0.0.0
  255.255.255.0  eth1;
132.100.101.1 132.100.100.0
  132.100.101.79 255.255.255.0 ;
132.101.27.34 132.101.27.0 external
  0.0.0.0 255.255.255.0 eth3;
}
// subnet *.100
 {
132.100.100.1  132.100.100.0  0.0.0.0
  255.255.255.0 eth1;
132.100.100.1 0.0.0.0 132.100.101.1
  255.255.255.0;
132.100.101.14 132.100.101.0 0.0.0.0
  255.255.255.0 eth0;
}
// subnet *.101
 {
132.100.101.1 132.100.101.0 0.0.0.0
  255.255.255.0 eth0;
132.100.100.27 132.100.100.0  0.0.0.0
  255.255.255.0 eth1;
132.100.99.34 132.100.99.0 0.0.0.0
  255.255.255.0 eth2;
132.100.101.79 0.0.0.0 132.100.99.1
  255.255.255.0;
}

// Response actions to check
// First response action
 {
 // Alternative 1-A
  { insertfwrule at 132.100.101.1:
     fw forward -destIP 132.100.100.0
     -destNm 255.255.255.0 -j deny;
 }

 // Alternative 1-B
 { insertfwrule at 132.100.99.1:
     fw forward -sourceIP  !132.100.0.0
     -sourceNm 255.255.0.0 -o eth2
     -j deny;
 }
}
 {  // Second response action
 // Alternative 2-A
 { insertfwrule at 132.100.99.1:
     fw forward -destIP 132.100.98.0
     -destNm 255.255.255.0 -j deny;
  }
}
```

Initially the response configuration does not contain any firewall rules (for the sake of simplicity). An ID system detects an attack coming from the Internet towards the machine 132.100.100.4, which is running the NFS-server. The IRS finds out that there are two ways to protect this server (labeled *alternative 1-A* and *alternative 1-B*). The IRS then requests the evaluation function to calculate the effects of both response actions. Alternative 1-A is inserted temporary into the model and the capability of all entities is determined, finding that it results in a reduced capability for the user Anne (because she will not be able to access the NFS server anymore) leading to a penalty cost of 2000. The capability for the entity Customer is not reduced because it can use one of two alternative HTTP servers. Even if one of them is not accessible the availability of the other one is sufficient and no additional penalty cost is caused by this response item. The total penalty cost for this alternative is therefore 2000.

The evaluation of alternative 1-B reveals that the capability of both, the customer and Anne are not reduced. User Customer is able to connect to the HTTP server in the 132.100.98.0 subnet, fulfilling its requirements. User Anne is not restricted as all required communication flow is allowed, resulting in a total penalty cost of 0.0. Through local optimization the IRS will clearly choose variant 1-B, because it has a lower overall penalty cost.

Afterwards the ID system detects another attack to the network 132.100.98.0/24, for which the IRS finds only one response action (namely alternative 2-A). As there are no alternatives to this response action, the local optimum choice can be determined easily. Unfortunately, together with alternative 1-B, the capability of the customer drops to 0.0, because the communication between this user and both HTTP servers is disallowed. This results in a total penalty cost of 100000 for this variant.

As can be seen easily, this is not desired, as the requirements of user customer cannot be met, which involves a very high penalty cost. When alternatives 1-A and 2-A would have been chosen, the total penalty cost would have been only 2000, keeping the overall use of the system as high as possible (clearly not for user Anne, but Customer is more important than she is). This emphasizes the importance of global optimization, as always choosing the local optimum does not lead to a globally optimized one. Notice that the reconfiguration will not change the security of the system, but increases the availability of important services.

Without our network model it was not transparent how an intrusion response system would react in the case of a detected intrusion. It might have even be the case that from the viewpoint of an attacker the behavior of the IRS is predictable. This is very dangerous, as the attacker could use the site's own IRS to perform a denial of service against that certain site by attacking in a special way. Response systems that do not know the mission of a network and that work on simplified models are vulnerable to this attack. On the other hand our network model enables the IRS to determine the real effects of countermeasures, finding the response configuration that provides protection against the attacks but in parallel keeping the usability of the system at its maximum level.

5.4 Implementation

We implemented a prototype of the network model and the evaluation function on Linux 2.4.18 using C. A parser to process the grammar for to specifying direct dependencies has been implemented using `flex` and bison.

We implemented the algorithms for capability calculation and for determining the overall penalty cost in an optimized way to keep the calculation complexity and the response required time of the system as low as possible. As much data as possible is pre-calculated to make the evaluation of response configurations efficient. The list of hosts that comprises the path between entities has to be iterated and the security policies of these hosts have to be tested against the exchanged packets.

When the system starts up the routing tables from all relevant routers of the network as well as the firewall rules are imported into the model at startup time. The configuration file containing the entity dependencies is read and transformed into internal data structures, i.e., it is merged with the network topology data.

For each host a list of its entities, the current routing tables as well as the firewall rules are kept. The paths of entities to entities they require to be fully functional are then pre-calculated. This can be done as we assume that the routing tables remain constant. Using the determined paths it is now possible to determine the individual indirect dependencies. For each involved entity we create a data structure that contains the following information:

- the current status of the running processes,
- links to the other entities that depend on the entity represented by this data structure, and
- management information, i.e., cost of being unavailable, current functionality and penalty cost.

This data structure allows the model to perform an efficient evaluation of response costs, as it is possible to perform an incremental evaluation of the total penalty cost. Instead of re-evaluating the model completely every time some response impact evaluation should be performed, only the parts of the model that might be affected have to be re-evaluated. For the other entities the penalty costs do not change.

When a response action is evaluated, all entities on the affected hosts have to be investigated. The functionality of those entities is recalculated. In the case that the new value is different from the old value, it is necessary to continue the process at the entities that depend on this one, otherwise evaluation can be aborted at this point. That should make clear that even when a large number of entities are used for the model, it is not necessary to perform a recalculation of the whole model. As response actions only affect a few points of the whole system, it is only necessary to trace the effects at these points.

Usually, the model is not re-evaluated completely (i.e., the capabilities for all entities are recalculated) when a new response is examined. Only when the system is initialized

or when the routing tables change, a complete re-evaluation has to take place, which is caused by the fact that the data-structures that hold the entities have changed.

As mentioned before, we currently support the update of firewall rules, the killing and restarting of processes and the disabling/enabling of user profiles at hosts. These are the most important long-term response actions and our network model can be utilized to calculate their impact. As our network model has to be exact, we also support special address translation modes SNAT and DNAT (source/destination network address translation) that are provided by firewalls such as iptables.

5.5 Evaluation

The presented model provides a way of determining the effects of firewall and process based intrusion responses. We proposed an evaluation mechanism that utilizes external information describing dependencies between resources in a network as well as their importance to different users to obtain an impact value for different response actions. Our model is in fact dynamic, as decisions are made on the current state of the network. It is flexible, as it can be used for different network topologies and can be reconfigured at runtime. We also saw that our model can keep the usability of a system as high as possible by performing a simulation of response effects before they are actually launched. Finally, we have to investigate he computational complexity of the evaluation model.

5.5.1 Theoretical Considerations

In order to evaluate the efficiency of this model, we have to investigate the different operations that are involved.

An optimized data structure is built during the initialization phase. Each resource contains information about the resources that it depends on their respective dependencies. The fixed evaluation order of the entities and the pre-calculated paths between entities allow a fast evaluation of the complete model.

The insertion and removal of a temporary response actions (the one which is currently examined) into the model is a crucial step because it has to be performed every time the impact of new a response action needs to be calculated. For the actual evaluation of the impact itself, the pre-calculated paths are utilized. This allows one to only take the effects of firewall rules and the availability of resources into account, making local optimization (i.e., finding the best response action among a set of alternatives) very efficient.

Finding a globally optimal response configuration is harder. It requires an exhaustive search of all possible combinations of alternatives in the *response history*, which is comprised by all response actions that have been suggested by the IRS. Although this search could be optimized too, it is still an expensive operation where the number of possibilities increases exponentially with the length of the history of the response actions that have been evaluated so far (shown in Figure 5.6).

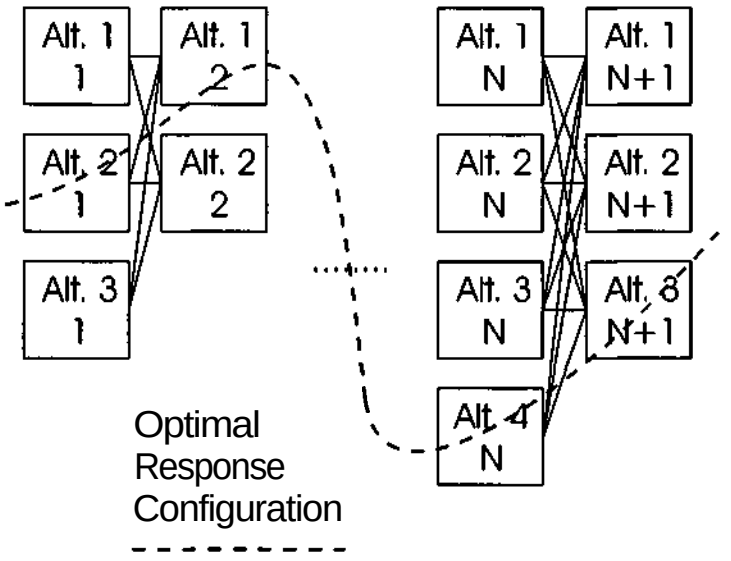


Figure 5.6: Response History and Globally Optimal Response Configuration

Nevertheless the individual alternatives are considered to be equivalent — they all protect against a certain attack. Despite the fact that local optimization can lead to non-optimal response configurations, the security of the system is guaranteed. So global optimization only enhances the usability of the overall system.

5.5.2 Performance Results

We obtained performance results of our prototype implementation to support the claims of our theoretical results. The execution times of different tasks performed by our prototype evaluation engine have been determined. We used a model with 35 resources which were heavily depended on each other — the dependency trees had a depth of up to eight. These resources were distributed over five subnets. We evaluated the impact of thirteen different response actions that consisted of up to ten firewall rule changes, user accounts and process status modifications.

The average number of alternatives in each test was 2.5384. The performance measurements (listed in Table 5.1 below) have been collected on a Pentium III machine with 550 MHz and 512 MB RAM. For each local optimization step, only up to eight alternatives had to be evaluated where we could make use of partial evaluation. The global evaluation, on the other hand, had to completely evaluate 5184 response configurations in order to determine an optimal solution.

The results show that evaluating different response actions can be done quickly. This is caused by the fact that only crucial resources are modeled and that optimized data structures are used during the evaluation process. While the complete entity capability evaluation is suitable for real time response, a complete global optimization may take longer, depending on the size of the response history and the number of alternative response

Insertion and deletion	0.0255ms
Complete entity capability evaluation	0.915ms
Global optimization step	34.358s

Table 5.1: Performance Results

actions. This is caused by the fact that many alternatives in a long history of response actions lead to an explosion of the number of combinations that have to be tested. While this seems to be undesirable at first glance, one has to realize that no real time performance is needed for this task. Even if the model requires a minute to find a globally optimal response configuration with an adequate level of security, the result is still beneficial. The security of the system has to be achieved first, then, in a second step, the usability of the overall system can be improved.

5.6 Summary and Conclusions

In this chapter we tackled the problem of automatic intrusion responses that have severe impact on the usability of a system. We identified the lack of a mission definition of network models as one major cause for this. We presented a network model that provides a method for specifying the mission of networks as well as an evaluation function that evaluates the impact of automated intrusion response mechanisms.

Our network model takes the network topology, firewall rules, services and users into account and supports both, dependencies among entities within the network and those to and from outside users. This enables the the model to determine the costs of disabling crucial resources due to responses. The evaluation mechanism which determines the negative impact exhibits good performance properties, especially the variant that determines the best action among a set of possible alternatives.

We built a prototype that can be used for determining *the* alternative that yields the minimal negative impact on deployed network resources and their users. The effects of 'severe' responses and their impact on the usability of the whole system can be estimated.

One problem remained open, which is the efficiency of global optimization. While choosing the best response action from a set of alternatives is very efficient, performing global optimization is very costly right now. Unfortunately no obvious mechanisms could be identified for making this process faster. Nevertheless global optimization is only an optional feature. The system is secured by launching suitable response actions, although the choice might not have been optimal with respect to the usability of a network. As the

system is in a safe state, more time can be used for finding a globally optimal response configuration.

Chapter 6

Conclusions and Future Work

> The future is much like the present, only longer.
>
> Dan Quisenberry

This chapter summarizes the results of this dissertation, evaluates the dissertation as a whole and points out ways where further research has to be performed. In the following we show the tackled problems and our contribution.

1. The first problem that is attacked by this dissertation is the performance of today's signature-based IDS, which is one of the most important ones as IDS have to be able to keep up with the event stream. The performance degradation was depicted for Snort, which uses optimized data structures during pattern matching. It was shown that high loads force ID systems to drop events. All other available systems provide only ad-hoc mechanisms for improving the detection speed, but have to compare an incoming event with all signatures. An analysis revealed that exactly this is the source of the performance problem — an incoming event is tested against each signature consecutively, yielding a $O(n)$ complexity for matching, where n is the number of used signatures. We proposed a decision tree based variant. The decision tree is created from the signatures using the ID3 clustering algorithm which uses the information gain as a heuristic for determining the feature that is used for splitting. Incoming events are used for navigating through the decision tree. Only the few signatures that matched all the conditions on the way through the decision trees have to be further investigated, i.e., the event's features that have not been used for decision tree construction have to be checked additionally. We optimized the individual feature comparisons by using customized data structures for each feature and applied a parallel string pattern matching algorithm for efficient string search. The experimental results show that using our decision tree based variant has a much more predictable timing behavior, making it much more resistant against attacks using worst case traffic. The additional initialization step for building decision trees, which is in the range of a minute, causes the ID system not to detect intrusions

directly after its invocation. It was shown that this is not a limitation, as ID systems are operated permanently. The memory usage has been in a range that can be tolerated on today's machines.

Future work has to deal with creating optimal decision trees, which again could improve performance. Unfortunately creating optimal decision trees is NP complete, but perhaps an optimized algorithm can be found to make this process efficient. In short the decision tree based approach is general and not constructed for a specific ID architecture — it has a high value for other signature-based ID system creators, as they can apply the gained results directly in their products.

2. The second problem that has been dealt with is the major shortcoming of being unable to detect unknown attacks, even if they belong to a known attack-class and are only variants. Current signature-based system get a number of signatures, which are accurate descriptions of events that contain intrusions. This makes it necessary to keep the signature database permanently updated to be able to detect the latest attacks. None of the existing signature-based systems are able to detect variants of attacks. We provided a way out of the permanent need to updating signatures by introducing the concept of abstract signatures. Abstract signatures capture the similarities of all instances of an attack-class and allow the ID system a detection of attacks belonging to the implemented attack-class, which also means that unknown attacks are detected. By setting several requirements on abstract signatures we can guarantee that the abstract signature will be useful in practice. We also presented a way of constructing abstract signatures in an efficient way. Two abstract signatures have been constructed, one for detecting buffer overflows and one for detecting worms. For each instance we set up a hypothesis which we proved either formally or by analyzing a huge amount of data. In both cases we showed that available exploits contain the similarity which does not show up in normal traffic. For detecting buffer overflow exploits we used the notion of maximum execution length to detect the sledge of a buffer overflow exploit in requests. We introduced the notions of connection chains and trails that are used for identifying causal connection patterns that represent a spreading worm. For both abstract signatures we implemented automated intrusion response mechanisms. In the case that a request containing a buffer overflow is detected, this request is simply dropped. If a spreading worm is detected, it is isolated by reconfiguring the personal firewalls. Vulnerable services are protected and the massive bandwidth consumption that is caused by the search activity of worms can be reduced if the system is widely deployed. Abstract signatures have a high utility, as they are update-free, have a very low false positive rate, can detect instances of a certain attack-class and are efficient to calculate.

   Future work has to concentrate on identifying and creating new abstract signatures. Unfortunately it is not guaranteed that for each attack-class an abstract signature can be found. But their advantages are overwhelming, and as already two instances of abstract signatures could be identified, more are likely to exist.

3. The third and last problem that has been tackled is making the use of automated response safe. It was shown that automated response is necessary and that existing methods for determining the response are insufficient, as they all work on oversimplified, static information. The actual cost of a response could not be determined, as the use/mission of a network was unspecified. To fight this problem, we introduced a network model that provides a facility for stating the mission of a network as well as the dependencies between the individual entities in the network. The network model combines the security policy, the network topology as well as the resource dependencies and transforms the result into an optimized data structure. The model allows the administrator of the response subsystem to specify to which degree an entity depends on another one and which cost is involved when a certain entity becomes partially or totally unavailable. We introduced the notion of *functionality* of resources and have been able to formulate an evaluation function that is able to calculate this given a certain scenario. Using the described elements of the model it is possible to determine the impact of several intrusion responses on the usability of the whole system. When an intrusion response system has to decide which alternative to pick from several, the model can evaluate the impact of each one and determine the one with the least impact on the usability of the whole system. The model is flexible in the sense that the real situation is reflected, as no simplifications are made and the current state is always used for determining the response cost. Choosing the best from a set of alternatives can be done very quickly. As our model always performs local optimization to meet its realtime requirements the reached states may be different from a global optimum one, which has the same security properties but has a lower total penalty cost. Because of the fact that the administrator only has to specify the direct dependencies of the entities the whole system is easy to administrate. Indirect dependencies are generated from the direct dependency specification as well as the security policy and the network topology, which have to be imported only once.

   Future work should extend the network model and the cost functions. Instead of deriving the capability of an entity from static dependency weights on various services, more sophisticated functions could be utilized. Usually penalty costs are not constants, but they are a function of time, and our model could be extended in this way. Work should also concentrate on improving the global optimization step, which is computationally expensive now, an exhaustive search has to be performed. Priority queues and dynamic programming might help in speeding up that process. Our model has been kept general and can be used for evaluating all types of responses. In the current model the importance levels of resources remained constant all the time. This might clearly not be optimal — at night times for example it might not be relevant that machines that are normally used for application development have access to the Internet, while an E-business application has to be reachable all the time. Therefore one could think of operating the response system in different modes, where different penalty costs are assigned to the same resources.

Bibliography

[1] Mechanisms to Implement Intrusion Response. http://www.sdsc.edu/DOCT/ Publications/e2/e2.html, August 1998.

[2] Antivirus approaches. www.cs.byu.edu/courses/cs565/slides/Ch9.pdf, January 2003.

[3] A. Aho and M. Corasick. Efficient String Matching: An Aid to Bibliographic Search. *Communications of the Association for Computing Machinery,* 18:333-340, 1975.

[4] Edward Amoroso. *Intrusion Detection - An Introduction to Internet Surveillance, Correlation, Trace Back, and Response.* Intrusion.Net Books, New Jersey, USA, 1999.

[5] D. Anderson, T. Frivold, A. Tamaru, and A. Valdes. *Next Generation Intrusion Detection Expert System (NIDES).* SRI International, 1994.

[6] M. Asaka, A. Taguchi, and S. Goto. The Implementation of IDA: An Intrusion Detection Agent System. In *11th FIRST Conference,* June 1999.

[7] Stefan Axelsson. Intrusion Detection Systems: A Survey and Taxonomy. Technical Report 99-15, Chalmers Univ., March 2000.

[8] C. A. Carver, J. M. D. Hill, and U. W. Pooch. Limiting Uncertainty in Intrusion Response. In *Proceedings of the 2001 IEEE Workshop on Information Assurance and Security,* United States Military Academy, West Point, June 2001.

[9] Curtis Carver. Intrusion Response Systems - A Survey, 2000.

[10] William R. Cheswick and Steven M. Bellovin. *Firewalls and Internet Security.* Addison-Wesley, Reading, Massachusetts, USA, 1994.

[11] Ids245 smtp-cmail-buffer-overflow. http://www.whitehats.com/info/IDS245, January 2001.

[12] F. Cohen. Simulating Cyber Attacks, Defenses, and Consequences. http://all.net/journal/ntb/simulate/simulate.html, May 1999.

[13] C. Jason Coit, Stuart Staniford, and Joseph McAlerney. Towards Faster String Matching for Intrusion Detection or Exceeding the Speed of Snort. In *Proceedings of DISCEX 2001,* 2001.

[14] Crispin Cowan, Calton Pu, David Maier, Heather Hinton, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Automatic Detection and Prevention of Buffer-Overflow Attacks. In *7th USENIX Security Symposium*, January 1998.

[15] Herve Debar, Marc Darcier, and Andreas Wespi. A Revised Taxonomy for Intrusion Detection. In *Annales des Telecommunications*, 2000.

[16] Marina del Rey. Transmission Control Protocol, DARPA Internet Program, Protocol Specification. http://www.faqs.org/rfcs/rfc793.html, 1981.

[17] K. Djahandari and D. Schnackenberg. Cooperative Intrusion Traceback and Response Architecture (CITRA). In *Proceedings of DARPA Information Survivability Conference and Exposition (DISCEX II'01) Volume 1*, June 2001.

[18] M. Ducass and e Pouzol. Handling Generic Intrusion Signatures is Not Trivial, 2000.

[19] S. T. Eckmann, G. Vigna, and R. A. Kemmerer. STATL: An Attack Language for State-based Intrusion Detection. In *ACM Workshop on Intrusion Detection Systems*, Athens, Greece, November 2000.

[20] L. Eschenauer. Imsafe. http://imsafe.sourceforge.net, 2001.

[21] M. Fisk and G. G. Varghese. An Analysis of Fast String Matching Applied to Content-Based Forwarding and Intrusion Detection. Technical Report UCSD TR CS2001-0670, UC San Diego, 2001.

[22] S. Forrest, S. A. Hofmeyr, A. Somayaj, and T. A. Longstaff. A Sense of Self for Unix Processes. In *IEEE Symposium on Research in Security and Privacy*, pages 120-128. IEEE Computer Society Press, 1996.

[23] Thomas D. Garvey and Teresa F. Lunt. Model-Based Intrusion Detection. In *Proceedings of the Fourteenth National Computer Security Conference*, October 1991.

[24] The GNU Compiler Collection. http://gcc.gnu.org.

[25] A. Ghosh and A. Schwartzbard. A Study in Using Neural Networks for Anomaly and Misuse Detection. In *USENIX Security Symposium*, 1999.

[26] Stephen Hansen and Todd Atkins. Automated System Monitoring and Notification with Swatch. In *Proceedings of the USENIX Systems Administration Conference (LISA '93)*, pages 145-152, 1993.

[27] Daniel Hartmeier. Design and Performance of the OpenBSD Stateful Packet Filter (pf). In *USENIX Annual Technical Conference - FREENIX Track*, 2002.

[28] Steve Hotz. DiG - Domain Name System query tool. http://www.centergate.com/shotz-pubs.html, 1990.

[29] Koral Ilgun. USTAT: A Real-Time Intrusion Detection System for UNIX. In *Proceedings of the 1993 IEEE Symposium on Research in Security and Privacy*, pages 16-28, Oakland, CA, 1993.

[30] Intel. *IA-32 Intel Architecture Software Developer's Manual Volume 1-3*, 2002. http://developer.intel.com/design/Pentium4/manuals/.

[31] ISS. BlackICE Overview. http://blackice.iss.net/.

[32] Peter Jackson. *Introduction to Expert Systems.* Addison-Wesley, Reading, Massachusetts, USA, 1986.

[33] K. Jensen. Coloured petri nets. In *Advances in Petri Nets 1986*, volume 1, pages 248-299, September 1986.

[34] K. Jensen. An Introduction to the Theoretical Aspects of Coloured Petri Nets. In *A Decade of Concurrency*, Lecture Notes in Computer Science 803, pages 230-272. Springer-Verlag, 1994.

[35] Y. Jou, F. Gong, C. Sargor, X. Wu, S. Wu, H. Chang, and F. Wang. Design and implementation of a scalable intrusion detection system for the protection of network infrastructure. In *DARPA Information Survivability Conference and Exposition (DISCEX'OO)*, pages 69-83, 2000.

[36] Home of K2. http://www.ktwo.ca.

[37] Calvin Ko. Logic Induction of Valid Behavior Specifications for Intrusion Detection. In *IEEE Symposium on Security and Privacy*. IEEE CS Press, May 2000.

[38] C. Krügel and T. Toth. Using Decision Trees to Improve Signature-Based Intrusion Detection. In *Recent Advances in Intrusion Detection*, October 2003. To appear.

[39] C. Kriigel, T. Toth, and C. Kerer. Decentralized Event Correlation for Intrusion Detection. In *International Conference on Information Security and Cryptology (ICISC)*. Lecture Notes in Computer Science 2288, Springer, December 2001.

[40] C. Kriigel, T. Toth, and E. Kirda. Sparta - A Security Policy Reinforcement Tool for Large Networks. In *IFIP Conference on Advances in Network and Distributed Systems Security*. Kluwer Academic Publishers, November 2001.

[41] C. Kriigel, T. Toth, and E. Kirda. Service Specific Anomaly Detection for Network Intrusion Detection. In *Symposium on Applied Computing (SAC)*. ACM Scientific Press, March 2002.

[42] Christopher Kriigel. *Network Alertness - Towards an adaptive, collaborating Intrusion Detection System*. PhD thesis, University of Technology, Vienna, 2002.

[43] S. Kumar and E. Spafford. A Pattern-Matching Model for Misuse Instrusion Detection. In *National Computer Security Conference,* October 1994.

[44] S. Kumar and E. Spafford. An Application of Pattern Matching in Intrusion Detection. Technical Report CSD-TR-94-013, Purdue University, West Lafayette, 1994.

[45] W. Lee, M. Miller, and S. Stolfo. Toward cost-sensitive modeling for intrusion detection, 2000.

[46] MIT Lincoln Labs. DARPA Intrusion Detection Evaluation. http://www.ll.mit.edu/IST/ideval, 1999.

[47] Wolfgang Lugmayer. *Gypsy: A Component-Oriented Mobile Agent System.* PhD thesis, University of Technology, Vienna, 1999.

[48] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX Winter,* pages 259-270, 1993.

[49] Todd McGuiness. SecureNet PRO Frequently Asked Questions. http://www.mimestar.com/products/faq.html, 2000.

[50] Todd McGuiness. Defense In Depth. http://www.sans.org/rr/securitybasics/ defense.php, November 2001.

[51] E. Moore. Cramming more components onto integrated circuits. Electronics 38)http://www.intel.com/research/silicon/moorespaper.pdf, April 1965.

[52] J.S. Moore and R.S. Boyer. A Fast String Searching Algorithm. *Communications of the Association for Computing Machinery,* 20:762-772, 1977.

[53] Mudge. Compromised: Buffer-Overflows, from Intel to SPARC Version 8. http://www.10pht.com, 1996.

[54] NetRanger. http://www.cisco.com/univercd/cc/td/ doc/product/iaabu/netrangr/, 2001.

[55] Network Flight Recorder. http://www.nfr.net/, 2001.

[56] Symantec Security Response, Nimda Analysis. http://securityresponse.symantec.com/ avcenter/venc/data/w32.nimda.a@mm.html.

[57] Peng Ning, Sushil Jajodia, and Xiaoyang Sean Wang. Abstraction-based intrusion detection in distributed environments. In *ACM Transactions on Information and System Security,* volume 4, pages 407-452, November 2001.

[58] V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. In *7th USENIX Security Symposium,* San Antonio, TX, USA, January 1998.

[59] P. A. Porras and P. G. Neumann. EMERALD: Event Monitoring Enabling Responses to Anomalous Live Disturbances. In *20th NIS Security Conference*, October 1997.

[60] Thomas H. Ptacek and Timothy N. Newsham. Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection. Technical report, 1998.

[61] J. R. Quinlan. Discovering rules by induction from large collections of examples. In *Expert Systems in the Micro-Electronic Age.* Edinburgh University Press, 1979.

[62] J. R. Quinlan. Induction of Decision Trees. *Machine Learning*, 1(1):81-106, 1986.

[63] M. J. Ranum, K. Landfield, M. Stolarchuk, M. Sienkiewicz, A. Lambeth, and E. Wall. Implementing A Generalized Tool For Network Monitoring. In *Eleventh Systems Administration Conference - LISA*, San Diego, California, USA, October 1997.

[64] Marcus J. Ranum, Kent Landfield, Mike Stolarchuk, Mark Sienkiewicz, Andrew Lambeth, and Eric Wall. Implementing A generalized tool for network monitoring. In *Proceedings of the Eleventh Systems Administration Conference (LISA '97)*, San Diego, CA, 1997.

[65] RealSecure. http://www.iss.net/customer_care/resource_ center/product_lit/, 2001.

[66] M. Roesch. Snort - Lightweight Intrusion Detection for Networks. In *USENIX Lisa 99*, pages 229-238, 1999.

[67] Mike Schulte. Cumputer Performance Examples and Cost. http://fizbin.eecs.lehigh.edu/ ~ mschulte/ece401-01/lect/my-lec02-p2.pdf, September 2001.

[68] SecurityFocus Corporate Site. http://www.securityfocus.com.

[69] S. R. Snapp, J. Brentano, G. V. Dias, T. L. Goan, L. T. Heberlein, C. Ho, K. N. Levitt, B. Mukherjee, S. E. Smaha, T. Grance, D. M. Teal, and D. Mansur. DIDS (Distributed Intrusion Detection System) - Motivation, Architecture and an early Prototype. In *14th National Security Conference*, pages 167-176, October 1991.

[70] Snort-NG. Snort - Next Generation: Network Intrusion Detection System. http://www.infosys.tuwien.ac.at/snort-ng.

[71] B. C. Soh and T. S. Dillon. Setting Optimal Intrusion-Detection Thresholds. *Computers and Security*, 14(7):621-631, 1995.

[72] E. H. Spafford. The Internet Worm Incident. In C. Ghezzi and J. A. McDermid, editors, *ESEC'89 2nd European Software Engineering Conference*, University of Warwick, Coventry, United Kingdom, 1989. Springer.

[73] S. Staniford, S. Cheung, R. Crawford, M. Dilger, J. Frank, J. Hoagland, K. Levitt, C. Wee, R. Yip, and D. Zerkle. GrIDS - A Graph Based Intrusion Detection System For Large Networks. In *20th National Information Systems Security Conference*, volume 1, pages 361-370, October 1996.

[74] D. Sterne, K. Djahandari, B. Wilson, B. Babson, D. Schnackenberg, H. Holliday, and T. Reid. Autonomie Response to Distributed Denial of Service Attacks. In *Proceedings of 4th International Symposium, RAID 2001*, Davis, CA, USA, October 2001.

[75] Aurobindo Sundaram. An introduction to intrusion detection. *Crossroads*, 2(4):3–7, 1996.

[76] Symantec. Symantec Host Intrusion Detection System. http://enterprisesecurity.symantec.com/products/products.cfm?ProductID=48&EID=0.

[77] tcpdump/libpcap. http://www.tcpdump.org.

[78] Touch Technologies. Touch Technologies, Inc. Product Offerings. http://www.ttinet.com/tti/nsa_www.html.

[79] H. S. Teng, K. Chen, and S. C. Lu. Security Audit Trail Analysis Using Inductively Generated Predictive Rules. In *Proceedings of the 6th Conference on Articial Intelligence Applications*, March 1990.

[80] T. Toth and C. Krügel. Accurate Buffer Overflow Detection via Abstract Payload Execution. In *Recent Advances in Intrusion Detection*, pages 274-291, October 2002.

[81] T. Toth and C. Krügel. Connection-history based anomaly detection. In *Proceedings of the 3rd Annual IEEE Information Assurance Workshop*, pages 241-246, June 2002.

[82] T. Toth and C. Krügel. Evaluating the Impact of Automated Intrusion Response Mechanisms. In *Proceedings of the 18th Annual Computer Security Applications Conference*, pages 301-310, December 2002.

[83] M. Undy. Tcpreplay. Software Package, May 1999.

[84] G. Vigna, S. Eckmann, and R. A. Kemmerer. The STAT Tool Suite. In *DISCEX 2000*, pages 1046-1053. IEEE Computer Society Press, January 2000.

[85] G. Vigna and R. A. Kemmerer. NetSTAT: A Network-based Intrusion Detection System. In *14th Annual Computer Security Applications Conference*, December 1998.

[86] WebSTONE - Mindcraft Corporate Site. http://www.mindcraft.com.

[87] G. B. White, E. A. Fisch, and U. W. Pooch. Cooperating Security Managers: A Peer-Based Intrusion Detection System. *IEEE Network*, pages 20-23, January/February 1996.

Curriculum Vitae

Thomas Franz Toth
Born: 12.02.1976, Vienna, Austria

Education

2000 - 2003	Ph.D. studies of computer science at Technical University Vienna. Teaching the compulsory lecture *Internet Security* at Technical University Vienna in Wintersemester 2001 and 2002.
Spring 2000	Graduation as Master of Science (Computer Science) at Technical University Vienna (with highest distinction)
1995 - 2000	Master studies of computer science at Technical University Vienna
1994-1995	Military service (compulsory)
Spring 1994	Graduation from high school (with highest distinction)
1986-1994	High school at Bundesrealgymnasium 22, Vienna, Austria
1982-1986	Elementary school at Volksschule 22, Vienna, Austria

Work Experience

2000 -2003	Project assistant at the Institute for Information Systems, Distributed Systems Group (E-1841). Working on the EU-funded intrusion detection related SPARTA-project (IST-12637).
July, August 1998 and 1999	Internship at Siemens PSE
August, September 1997	Internship at Management Data
July 1997	Internship at Bank Austria
July 1996	Internship at Waagner-Biro
May 1995 - July 1995	Internship at Waagner-Biro
September 1994	Internship at Erste Bank
August 1994	Internship at Waagner-Biro

Research Interests

- Intrusion detection and intrusion response
- Network and Computer Security
- Distributed Systems

Publications

http://www.infosys.tuwien.ac.at/Staff/tt